GROSS MOTION PLANNING OF AN OBJECT THROUGH A MAZE OF OBSTACLES

by

G. SRINIVASARAGHAVAN

DEPARTMENT OF MECHANICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JULY, 1986

GROSS MOTION PLANNING OF AN OBJECT THROUGH A MAZE OF OBSTACLES

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

by

G. SRINIVASARAGHAVAN

to the

DEPARTMENT OF MECHANICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JULY, 1986

CERTIFICATE

This is to certify that the present work GROSS MOTION PLANNING OF AN OBJECT THROUGH A MAZE OF OBSTACLES, has been carried out by Mr. G. Srinivasaraghavan under our supervision and it has not been submitted elsewhere for a degree.

(H. Hatwal)
Assistant Professor

(A. K. Mallik)
Professor

Department of Mechanical Engineering
Indian Institute of Technology
Kanpur 208 016
INDIA

July, 1986

CERTIFICATE

This is to certify that the present work GROSS MOTION PLANNING OF AN OBJECT THROUGH A MAZE OF OBSTACLES, has been carried out by Mr. G. Srinivasaraghavan under our supervision and it has not been submitted elsewhere for a degree.

Hatwal

Amallik

(H. Hatwal) Assistant Professor

(A. K. Mallik) Professor

Department of Mechanical Engineering
Indian Institute of Technology
Kanpur 208 016
INDIA

July, 1986

ACKNOWLEDGEMENTS

I express my heartfelt thanks to Dr. A.K. Mallik and Dr. H. Hatwal, under whose guidance, I did this work. I got all the encouragement that one could hope for, from them, throughout. I was given enough independence to try out my ideas though I was checked when I was on the wrong path.

I wish to thank Dr. S.G. Dhande for having given me access to the wonderful facilities available for Computer Aided Design, Graphics etc. in the CAD Centre.

My thanks are also due to Swami Anand Chaitanya for having done an efficient job of typing this thesis out for me.

I am greatly indebted to all my friends in Hall V. They are a crazy lot, nevertheless, they made my stay here a pleasant one.

July, 1986 G. Srinivasaraghavan

CONTENTS

LIST OF FIGURES

SYNOPSIS

The present work, proposes algorithms for finding collision free paths for a rigid polygonal or a circular object moving throu space that is cluttered with polygonal obstacles. The path can include rotations of the object. Two algorithms, one based on the Penalty Function Approach, and the other based on the configuration space approach are have been proposed.

The algorithm based on the Penalty Function Approach, is an improvement on those proposed earlier by researchers, in that (i) it calculates the path directly in terms of the joint coordinates of the robot under consideration and (ii) it takes into account the exact shapes of the object and the obstacles into consideration for computing the penalty for nearing the obstacles.

The algorithm based on the configuration space approach, uses a more efficient algorithm for searching through the free space for a path, than the earlier algorithms proposed by others. This algorithm works by starting with a straight line path between the end points and then recursively modifying it to finally arrive at a collision free path.

CHAPTER I

INTRODUCTION

1.1 Introduction

Robots are destined to play a significant role in the industrial scene of the future. This is primarily because they can be applied to a large variety of tasks. Most of the tasks which are potentially dangerous or monotonous, done earlier by human beings are now being taken over by robots.

Most of the tasks done by a human operator, need a certain amount of 'skill' achieved by conscious or subconscious training. If robots are to perform similar tasks, some decision making capability is essential and this is provided by a computer. So most of the problems associated with imparting 'skill' to the robots, reduce to that of developing suitable algorithms that can be implemented on a computer. The present work aims at developing algorithms for what is known as the 'Findpath Problem'. This problem can be stated as: given an <u>object</u> with an initial position (location and orientation), a final position and a set of <u>obstacles</u>, located in space, find a continuous path for the object from the initial to the final position, that avoids collisions with all the obstacles. The Findpath Problem is associated with gross motion

planning and is the first problem to be solved in overall task planning. The other planning aspects, namely grasping planning and finemotion planning are not considered here.

1.2 Review of Previous Work

Most of the algorithms proposed earlier, for solving the Findpath Problem, fall under the following three groups [1]:

1. Hypothesize and test method
2. Penalty function method
3. Explicit free space method

1. Hypothesize and Test Method:

This was the earliest approach tried, towards solving the Findpath Problem. As the name itself suggests, the method is based on first proposing a path for the object from the initial to the final position, and then testing the path to see if it avoids collisions with all the obstacles. If it does not, then, the path is modified to avoid the collisions. This is done repeatedly till the path obtained is free of all collisions. So the method basically relies on algorithms for (i) detecting possible collisions of an object, moving along a given path, with the obstacles and (ii) modifying an unsafe path to yield one which is safe. Most of the algorithms (or heuristics) proposed [2] for the modification of an unsafe path are based on approximations of the shapes of the object and the obstacles into regular geometric shapes. These approximations are often drastic and hence methods based on

these fail to work, when the workspace is closely cluttered with obstacles. However, if the obstacles are sparsely located, these methods work quite efficiently because of their inherent simplicity.

2. Penalty Function Methods

These methods are based on proposing a function of the position of the object, which when minimized, leads to the destination. The function value increases sharply as the object nears an obstacle , approaching infinity when the object collides with it. This provides a repulsive force which keeps the object away from the obstacle. In case there are many obstacles, then the penalties for nearing each one of them, are just added up to yield the resultant penalty.

The minimization of the function could be done using the partial derivatives of the function with respect to the configuration parameters. The sequence of values of the configuration parameters, obtained during minimization, represents the sequence of positions of the object from its initial to final position and in turn the path of the object.

The major drawbacks of the penalty function method are:
(i) Most of these methods [3] use drastic approximations of the object and the obstacles with regular shapes such as circle square etc. or in the case of 3-D, by spheres, cylinders, cubes etc. This renders it unreliable in many cases, and in some others, paths which are actually feasible, are ruled out.

(ii) All the minimization methods guarantee convergence only to a local minimum because of the strictly local information that is made use of. Hence if there are minima (local) other than the destination, minimization might lead to one of these other minima. In such cases, since a minimum has been reached, no further progress can be made. So the algorithm will have to back track to one of the earlier configurations and resume the search in a direction different from the one followed earlier. But identifying the points from where the search is to be resumed, might be very difficult.

3. Explicit Free Space Methods

These methods are based on building explicit representations of those configurations of the robot that are free of collisions. The set of these collision free configurations is also refer to as the Free Space . The Findpath Problem in this case is that of finding a path through the free space for the object. The different algorithms proposed, differ primarily in their characterization of the free space and the particular subsets of the free space they consider for finding a path. Lozano-Perez [4] suggested an algorithm which represents the free space by thecomplement of the space occupied by the obstacles, which have been expanded so as to allow the object to be shrunk to a point. A reference point is chosen for the object. This is the point to which the object is shrunk. The algorithm, then searches for a path for this reference point through the free space. To do this, the free

space is broken up into polyhedral cells of varying resolutions and then a graph, whose nodes denote these cells and the edges represent the overlap between the cells, is formed. A path is found by searching through this graph. If the object to be moved is not convex, it is broken up into a union of convex polygons and then the above algorithm is applied.

Fig. 1.1(a) shows an object 'O', which is a union of two convex polygons P and Q, moving from a position O to position O'', without rotations. 'R' is its reference point and O_1, O_2 are the obstacles on the way. Fig. 1.1(b) shows how this problem gets transformed to one where the point R is to be moved from R to R'' with the obstacles O_{1q}, O_{1p}, O_{2q} and O_{2p} on the way. Here O_{ip} and O_{iq} are the polygons obtained after expanding the obstacle O_i to account for the polygons P and Q respectively. In case rotation is also allowed, a third dimension gets added to problem in 2-D. For a general motion in 3-D space, the configuration space becomes six dimensional.

Another way of representing the free space was suggested by R.A. Brooks [5]. Here, the free space is represented at a union of generalized cones. These cones are in the form of corridors or pathways between adjacent obstacles. Valid orientations of the object as it moves through a generalized cone are first determined and then a path, composed of segments of the axes of these cones is looked for. The generalized cones generated for a set of obstacles is shown in Fig. 1.2.

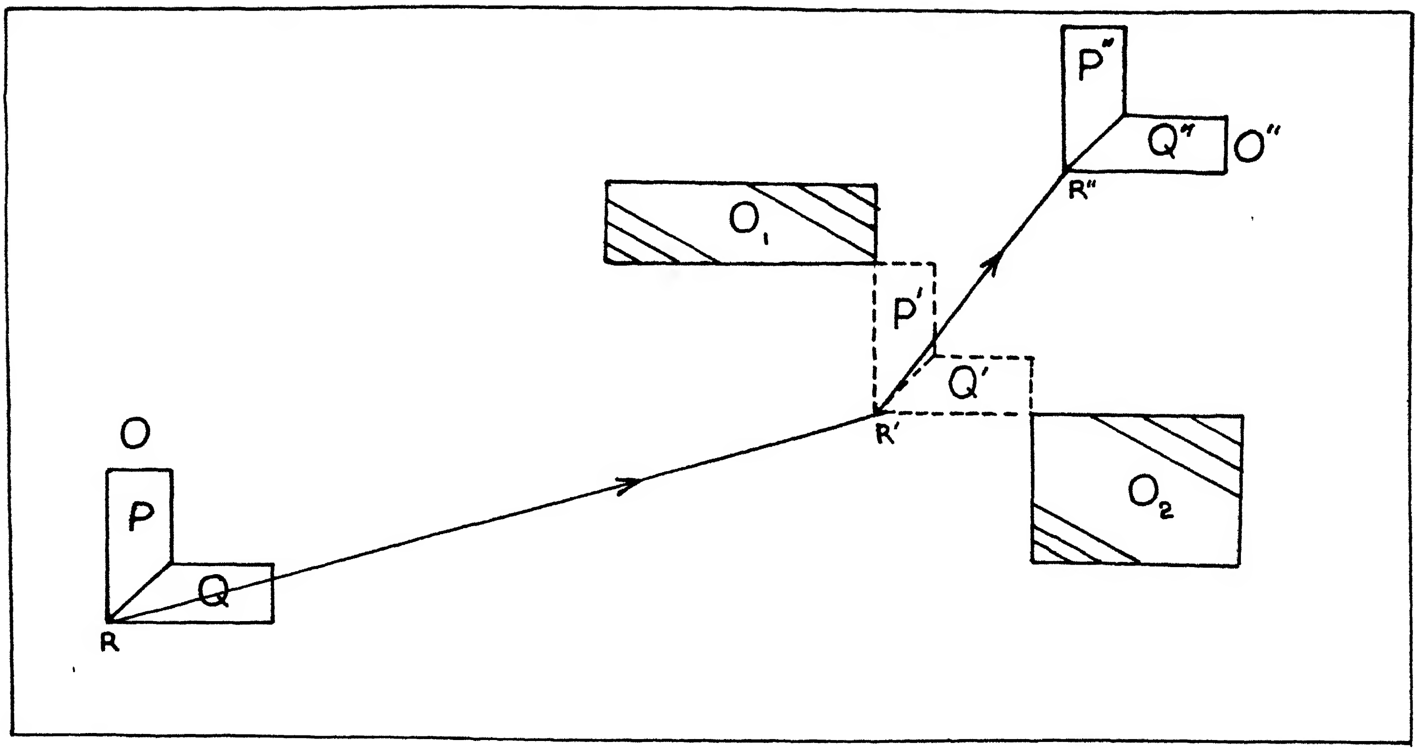


Fig.1.1(a)

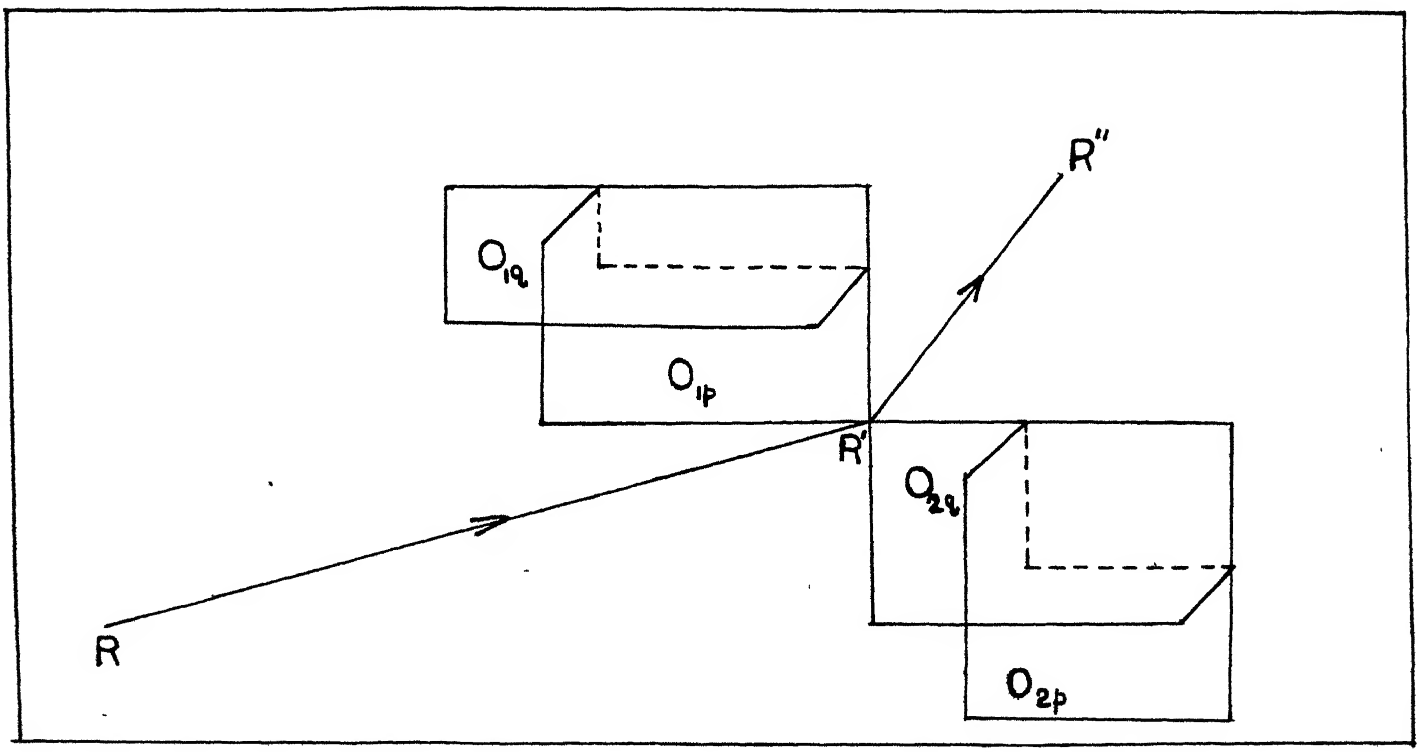


Fig.1.1(b)

An Illustration of the Configuration Space Approach

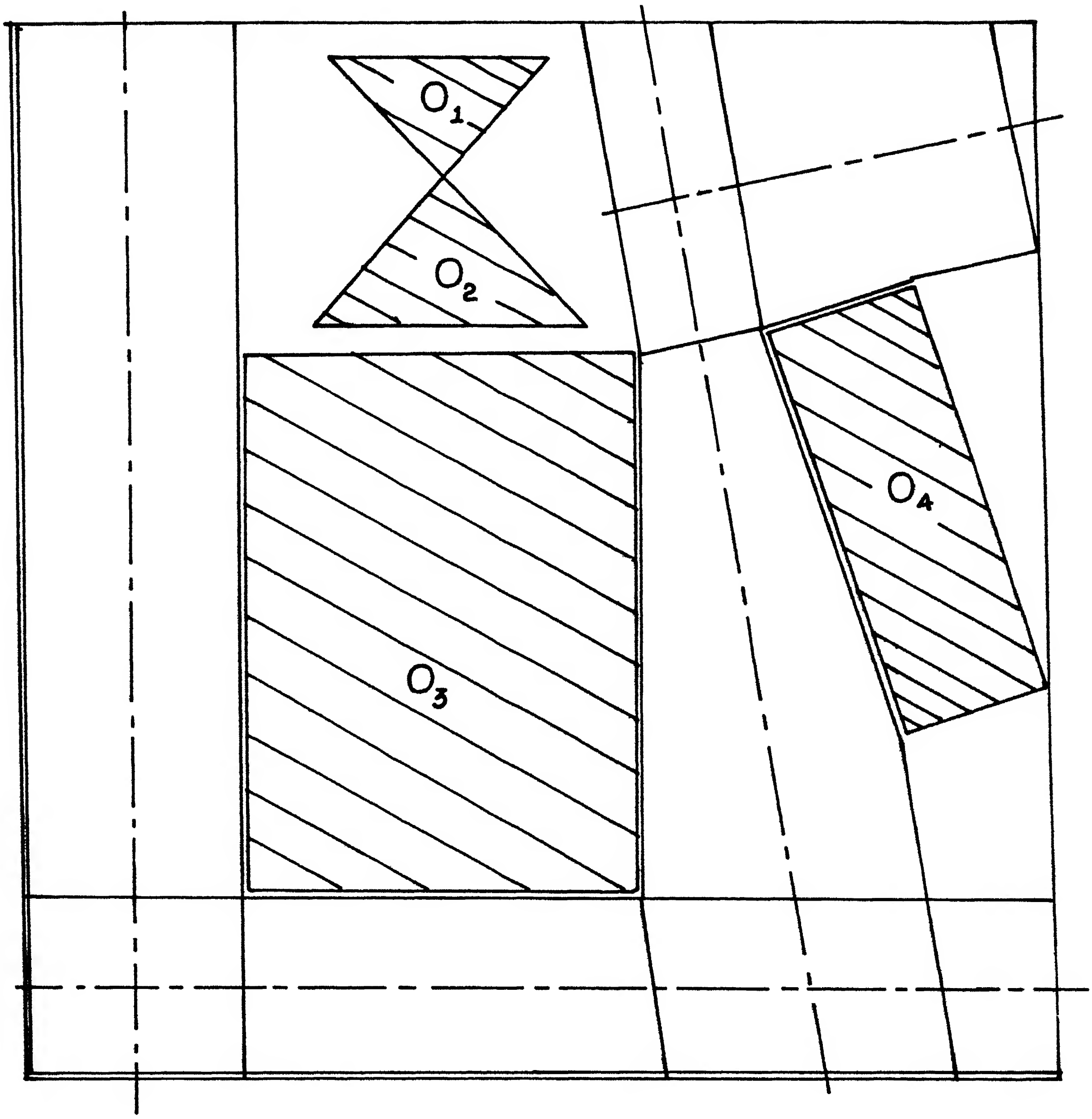


Fig.1.2 Generalized Cones for a Set of Obstacles

The major attraction of thefree space methods is that, these algorithms will find a path, if one exists. Moreover, instead of just looking for a feasible path, some kind of optimization can also be incorporated allowing one to look for the shortest or the least expensive path. The disadvantage is that the computation of the free space might be very expensive. However, these algorithms work very well, even when the obstacles are closely cluttered. For such closely cluttered workspaces, other algorithms either fail or spend an undue amount of effort in path searching.

Most of the algorithms that have been proposed for obstacle avoidance, are fundamentally tied to the use of object approximations. These are inapplicable in cases where the object has to move very close to the obstacles. The algorithm suggested by Lozano-Perez and R.A. Brooks [6] is very good in this respect though it is limited to cartesian robots and is also computationally expensive. An efficient obstacle avoidance algorithm for general robots with revolute joints remains to be developed.

1.3 Scope of the Present Work:

The present work proposes two algorithms, towards solving the Findpath Problem. The first algorithm is based on the penalty function approach. It incorporates the following features:

(i) The path is generated directly in terms of the joint coordinates, thus avoiding computations to be carried out for the inverse transformation from cartesian to joint space.

(ii) Unlike previous work, using the penalty function approach, the proposed algorithm takes into account, the exact shapes of the object and the obstacle concerned.

The second algorithm is based on the configuration space approach suggested by Lozano-Perez [4]. The subdivision algorithm suggested by Lozano-Perez and R.A. Brooks [6] for searching through the configuration space, divides the whole free space into polyhedral cells and then searches for a path through these. This involves, more computation than what is actually necessary because the cells which are most significant are the ones close to obstacles. The algorithm proposed here, though is not based on subdivision of the free space, concentrates more on the neighbourhoods of the obstacles. This is expected to reduce the amount of computation needed and also the storage space required for the algorithm to be run.

CHAPTER II

FORMULATION OF THE ALGORITHMS

2.1 Introduction

This chapter gives detailed descriptions of the algorithms developed along the lines sketched in Section 1.2. The algorithm based on the Penalty Function Approach, is presented in Section 2.2. Subsequently, the algorithm based on the Configuration Space Approach is discussed in Section 2.3.

2.2 Penalty Function Approach

This section describes a procedure to solve the findpath problem, using the Penalty Function Approach . Algorithms for calculating the penalties for nearing an obstacle, that take into account the exact shapes of the bodies concerned, are first presented. A penalty function method, that makes use of these algorithms, is then formulated.

When an object is being moved through a maze of obstacles, without colliding with any of them, penalties are imposed for nearing any of the obstacles. Closer the object gets to an obstacle, the higher is the penalty it receives. So the penalty imposed depends essentially on the nearness of the object to an obstacle. For this, a way of quantifying the nearness or closeness of an object to an obstacle needs to be found.

Before attempting to develop algorithms for doing this, we first define the 'nearness' between two bodies. Let O_1 and O_2 be the sets of points that constitute the two bodies. Let (X, Y) be the pair of points such that $X \varepsilon O_1$ and $Y \varepsilon O_2$, for which $||X - Y||$ (distance between the two) is a minimum. This distance denoted by d_{min}, gives the smallest distance of separation or the 'nearness' between the two bodies. Algorithms for computing the d_{min} for any pair of bodies, are now proposed.

We confine ourselves to problems on 2-dimensions, i.e. problems which involve movement of a planar object, through a maze of obstacles which are also planar. We also assume that the object and the obstacles are either circles, or arbitrary polygons. With these, the simplest case that one can think of is that of a circular object approaching an obstacle, which is also circular. In this case, d_{min} is given by $d_{min} = d_c - r_1 - r_2$, where r_1 and r_2 are the radii of the object and the obstacle, respectively and d_c is the distance between the two centres.

A problem, slightly more complicated than this is one where a circle approaches a polygon.

A circle can hit a polygon in one of the following two ways: (i) One of the vertices of polygon could hit the circle or (ii) the circle could touch the polygon along one of the edges. These two cases have been illustrated in Fig. 2.1(a).

The algorithm, that takes into account both these cases, is described below. Let O_1 and O_2 be the circle and the polygon

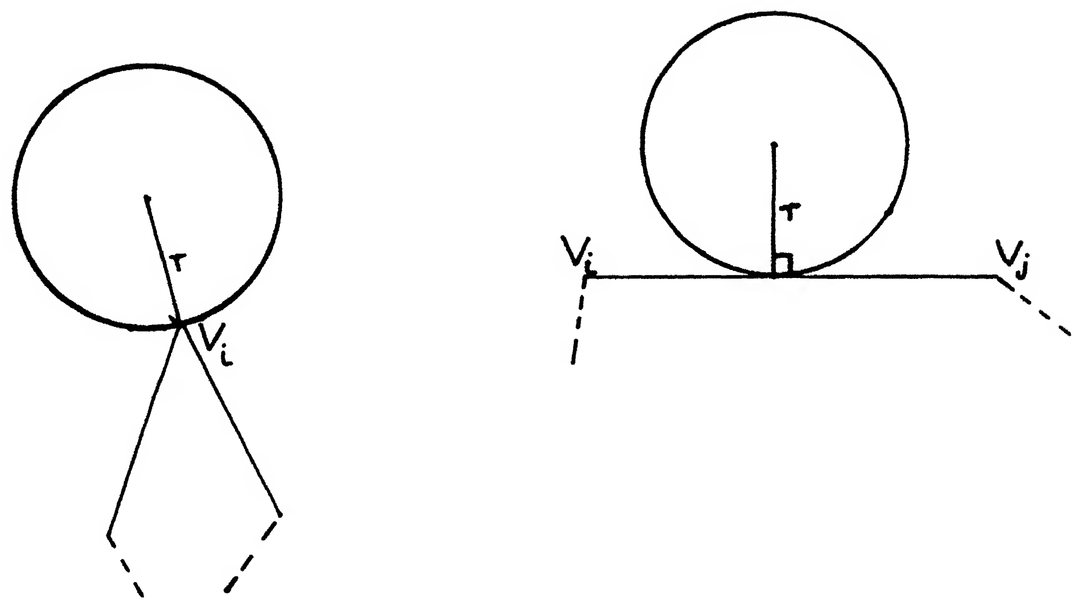


Fig. 2.1(a) Two Ways in which a Circle and a Polygon can come into contact.

respectively, approaching each other, with R being the centre of the circle, as shown in Fig. 2.1(b). A vertex V_i of the polygon, is first chosen. The distances dV_i and dE_i of the point R from the vertex V_i and from the edge V_iV_{i+1} (clockwise edge starting from V_i, with respect to a point within the polygon), respectively, are computed. A check is now made to see if the foot of the perpendicular from R to $\overline{V_iV_{i+1}}$ falls within the edge, i.e., between V_i and V_{i+1}. If it is not so, then dE_i is assigned an arbitrarily large value (as for dE_1 in Fig. 2.1(b)). This is done because, when the circle hits an edge of the polygon, the edge becomes tangential to the circle with the point of tangency being obviously within the edge. The line joining the centre of the circle to the point of tangency, will then be perpendicular to the edge. Hence if the foot of the perpendicular, dropped on an edge of a polygon, from the centre of a circle, falls outside the edge, then the edge is definitely not a potential fouling edge.

The dE_i's and dV_i's are computed for every i, i = 1,n where n is the number of vertices of the polygon. Let dE be the smallest of the dE_i's and let dV be the smallest of the dV_i's. Now the smaller of the two, dE and dV, becomes the d_{min} for this pair (O_1 and O_2) of bodies.

The most general problem is that of a polygon approaching a polygon. We first consider only convex polygons. The method

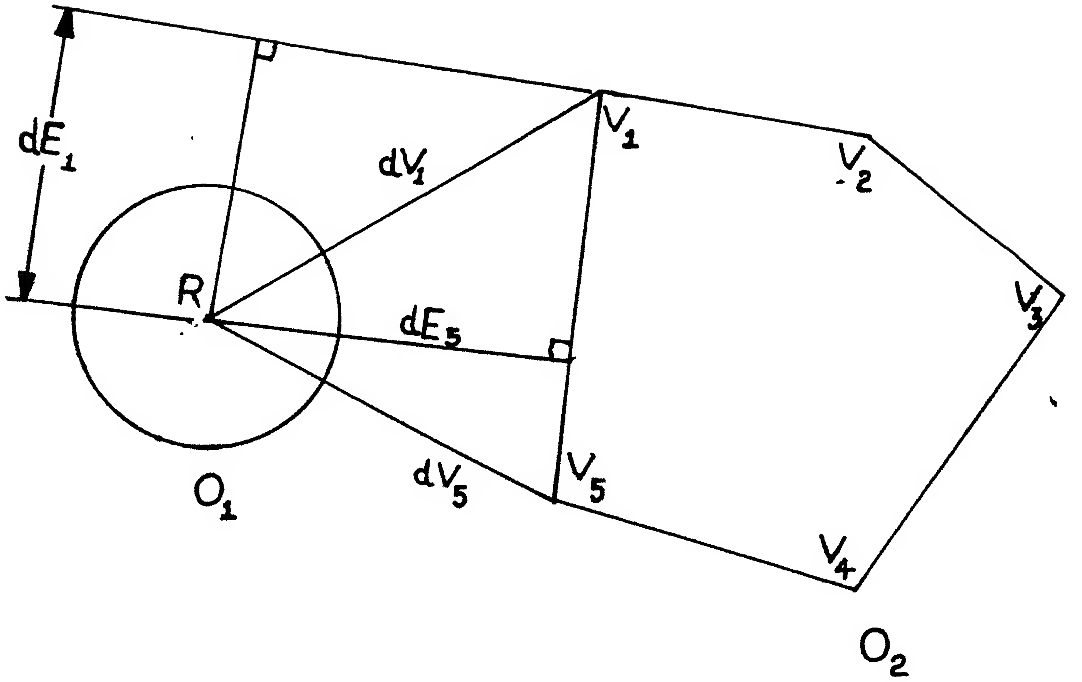


Fig. 2.1(b) Determination of d_{min} between a Circle and a Polygon

developed here will later be extended to non-convex polygons as well. Here, again, we have two possible ways in which a collision could occur between a pair of polygonal bodies -
(i) a vertex of one polygon hitting a vertex of the other,
(ii) a vertex of one, touching an edge of the other. An algorithm, which takes into account both these, is described below.

In Fig. 2.2, let O_1 and O_2 be the two polygons, under consideration, with R_1 and R_2 being their respective centres (these centres can be a rough estimate of the exact centres). The R_i's ($i = 1,2$), can be located as points with coordinates given by,

$$\left(\frac{x^i_{max} + x^i_{min}}{2} , \quad \frac{y^i_{max} + y^i_{min}}{2} \right)$$

where x^i_{max} is the largest of the x-coordinates of the vertices of O_i, and x^i_{min}, the smallest. Similarly, y^i_{max} and y^i_{min} refer to the maximum and minimum values, respectively, of the y-coordinates of the vertices.

Having located R_1 and R_2, the segment R_1R_2 is drawn and this line segment is checked for intersections with the edges of O_1 and O_2. Let $V_1^i\ V_1^j$ (say E_1) and $V_2^p\ V_2^q$ (say E_2) be the edges of O_1 and O_2 respectively, that intersect R_1R_2 (in Fig. 2.2, $E_1 = V_1^3\ V_1^4$, and $E_2 = V_2^5\ V_2^6$). The distances of the two end-vertices of E_1 from E_2 and those of E_2 from E_1 are first determined. Here, while trying to compute the distance of a vertex of one polygon

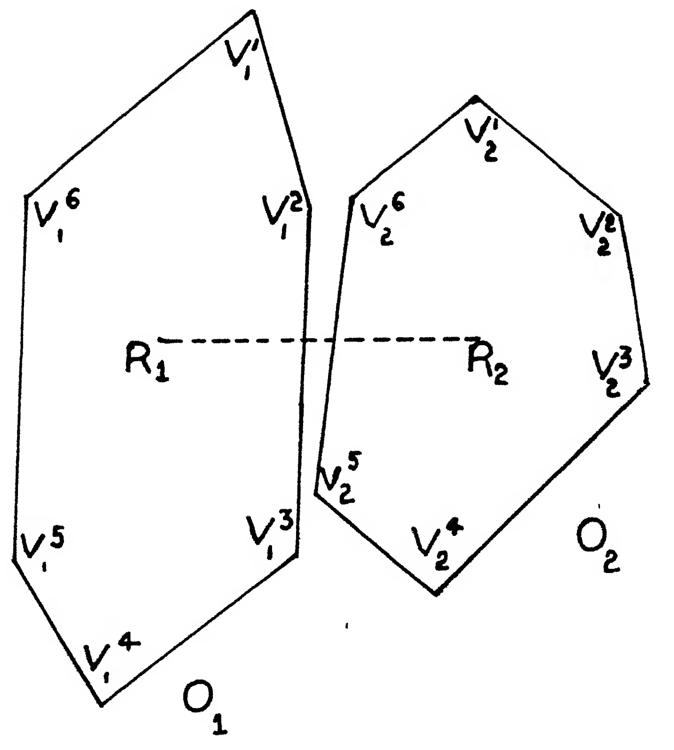


Fig. 2.2 Determination of d_{min} between Two Polygons

from an edge of the other, a test is made to ensure that the foot of the perpendicular from the vertex onto the edge falls within the edge. If this does not happen for any of the four distances (V_1^i and V_1^j from E_2 and V_2^p and V_2^q from E_1), then d_{min} is simply the minimum of the distances between pairs of vertices, i.e. the minimum of $|\overline{V_1^i V_2^p}|$, $|\overline{V_1^i V_2^q}|$, $|\overline{V_1^j V_2^p}|$ and $|\overline{V_1^j V_2^q}|$. Otherwise, the minimum of the four vertex to edge distances, subject to the above mentioned condition, is found. Let this distance be d with the corresponding vertex and edge being V_1^i and E_2 respectively. With these as initial data, a search is now carried out to determine the d_{min}.

The distance d of every successive vertex from V_1^i along the direction $\overrightarrow{V_1^j V_1^i}$, from the edge E_2 is computed and d updated everytime until d stops decreasing or the foot of the perpendicular from a vertex falls outside E_2. Let the last vertex, that satisfies these criteria be V_1^k and let the edge incident on V_1^k in the direction of search ($\overrightarrow{V_1^j V_1^i}$) be $\overline{V_1^k V_1^l}$. Let V_2^p be the end vertex of E_2, in the direction opposite to that of $\overrightarrow{V_1^j V_1^i}$, (i.e. if $\overrightarrow{V_1^j V_1^i}$ is clockwise, this direction is taken as anti-clockwise) where the directions are all with respect to a point within the corresponding polygon. A similar search is now carried out, starting with the present value of d, the vertex V_2^p and the edge $\overline{V_1^k V_1^l}$.

This whole process is repeated until d can not be decreased further. The value of d thus obtained becomes d_{min} between O_1 and O_2.

The above algorithms are quite efficient, in the sense that their complexities are not of a very high order. In the case involving a circle and a polygon, as is evident from the procedure outlined, the search involves going round the list of vertices of the polygon, just once, i.e. is of complexity $O(n)$ where n is the number of vertices of the polygon. For the case involving two-polygons, the worst case complexity would be $O(n+m-4)$ where m and n are the number of sides of the two polygons. The average complexity would be much less because, using the line $\overline{R_1R_2}$ to start the search serves as a very good heuristic and it shortens the search considerably.

In case, one of the two polygons is not convex, the non-convex polygon is broken up into parts, each of which is a convex polygon, i.e. if O_1 is a non-convex, planar body, then we split it up as,

$$O_1 = \bigcup_{j=1,N_c} O^j_{c1}$$

where O^j_{c1} is the j-th convex component of the non-convex polygon O_1 and N_c is the number of components. Assuming O_2 is the other polygon (d_{min} is to be computed for O_1 and O_2), d^j_{min}, which is the closest distance of approach between O^j_{c1} and O_2, is computed for all $j = 1, N_c$. The minimum of all these d^j_{min}'s gives the d_{min} for O_1 and O_2.

We now formulate an obstacle avoidance algorithm, based on the penalty function approach, that makes use of the algorithms given above. Any such algorithm, essentially produces a sequence of positions of the object to be moved, which defines the path that avoids collisions with the obstacles on the way. This sequence of positions can either be generated in terms of the cartesian coordinates or directly in terms of the joint-coordinates of the robot being used to execute the motion. Since, any motion, tobe executed by a robot, will ultimately have to be defined in terms of its joint-coordinates, a path defined in terms of the cartesian coordinates will have to be transformed to an equivalent path in the joint-coordinates, for it to be executed. This transformation, from the cartesian frame to the frame of the joint-coordinates, is computationally very expensive. Generating a path directly in the joint-space, is although not as easy a task as doing it in the cartesian-space, it avoids the inverse-transformation to be carried out at every point along thecartesian-path if the path is defined in the cartesian-space. Here we take the latter option, i.e. of generating a path directly in terms of the joint co-ordinates of the robot.

The starting and the goal positions of the object to be moved are first defined. This definition will obviously be in terms of the cartesian coordinates because a position defined in terms of the joint-coordinates cannot possibly be visualized.

The representation we adopt for this definition is as follows.

As shown in Fig. 2.3, a reference point is chosen for the object. If the object is a circle, its centre is chosen as its reference point $R(x_o, y_o)$ and if it is a polygon, some arbitrary point (preferably one close to its intuitive centre) inside the polygon is chosen $R'(x_o', y_o')$ as the reference point. A local frame of reference, with its origin at the reference point is then chosen. This local frame is fixed rigidly to the object. The object is now described in terms of these local coordinates. If the object is a circle (O with the local frame $x_\ell - y_\ell$), this description would simply be the radius of the circle. If the object is a polygon (O with the local frame $x_\ell' - y_\ell'$), specifying the local coordinates of the vertices of the polygon, would be a complete description of the polygon. So, the position of an object can be completely defined by specifying the local frame of the object with respect to the global frame X-Y. In case the object is a polygon, apart from the coordinates of the origin of the local frame, the orientation of the local frame with respect to X-Y will also have to be specified. This angle is denoted by ϕ.. As in Fig.2.3, the position of a circular object can therefore be defined as (x_o, y_o) i.e. thecoordinates of the origin of $x - y$ and that of a polygonal object as (x_o', y_o', ϕ).

To generate the path, directly in terms of the joint-coordinates, the starting and the goal positions of the object,

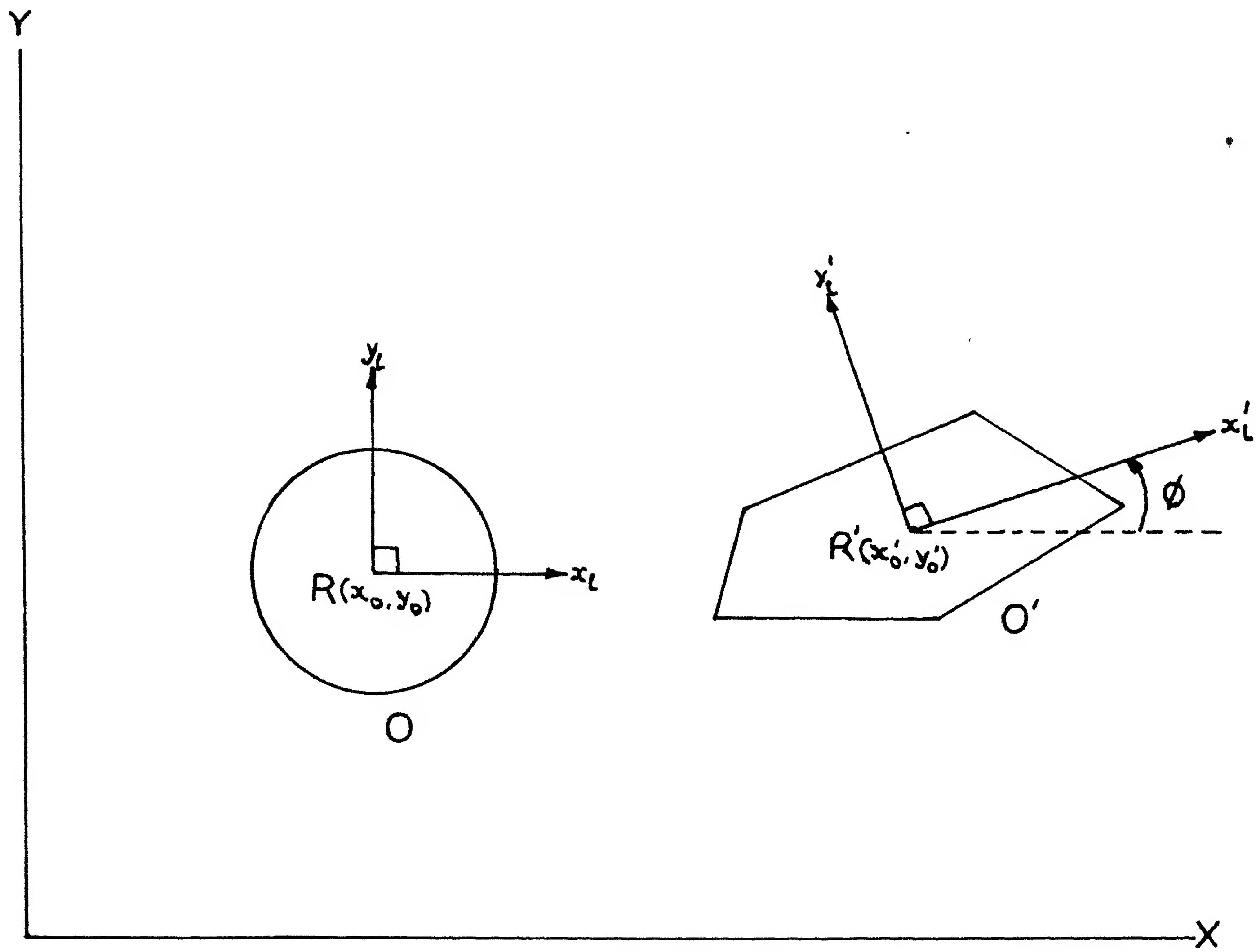


Fig.2.3 Representational Scheme for the Object and the Obstacles

defined in the cartesian space, are first transformed to their equivalent joint-coordinates. This transformation from cartesian-space to the joint-space, is specific to a particular mechanism and can in no way be generalised. The specific mechanism considered here is a 4-degree of freedom (5 degrees of freedom if additional wrist roll for non-circular objects is included) robot, the configuration of which is given in Fig. 2.4. The procedure adopted here for this transformation is due to Paul [7].

Let the joint-angles be θ_1, θ_2, θ_3, θ_4 (and θ_5 for the additional rotation) and let the link lengths be ℓ_1, ℓ_2, ℓ_3 and ℓ_4. The axes for the rotations θ_2, θ_3 and θ_4 are all normal to the plane of the mechanism and that for θ_1 is parallel to the Z-axis. The axis of rotation for θ_5 is along the link 4. The motion takes place on the x-y plane.

Let the local frames of reference for each one of the links of the robot be as shown in Fig. 2.5. The object is held such that the origin of the frame attached to the end effector $(x_4 - y_4 - z_4)$ coincides with the reference point of the object. Now, if $[T_i]$ is the matrix description of the frame $x_i - y_i - z_i$ with respect to the frame $x_{i-1} - y_{i-1} - z_{i-1}$, we have

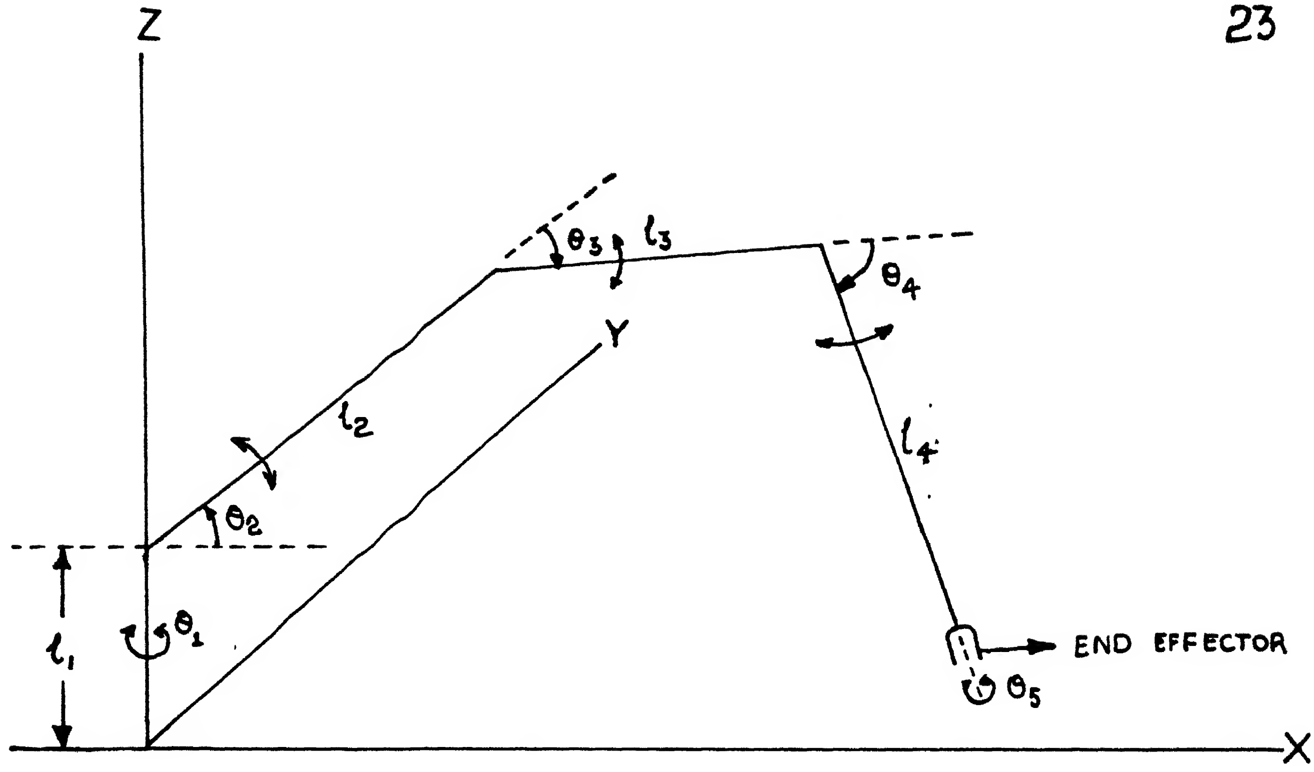


Fig. 2.4 Configuration of the Manipulator

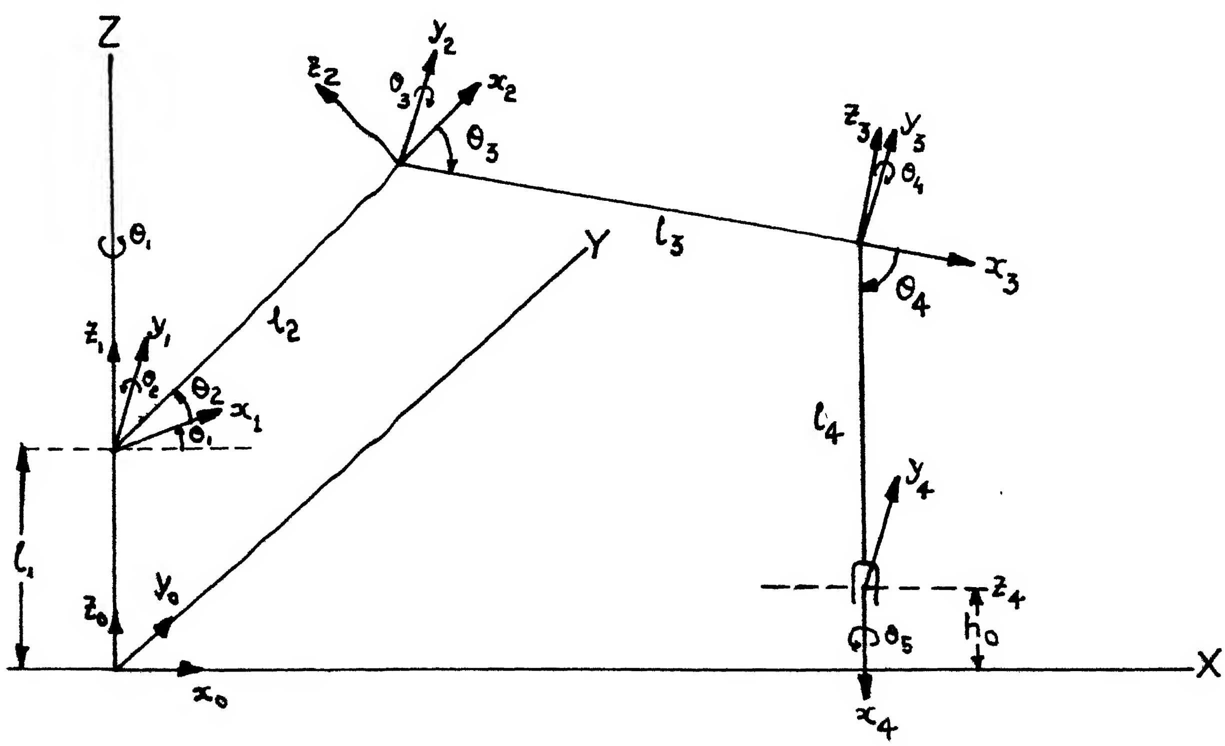


Fig. 2.5 Local Frames of reference for the Links of the Manipulator

$$[T_1] = \begin{bmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & 1 & \ell_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad [T_2] = \begin{bmatrix} c_2 & 0 & -s_2 & \ell_2 c_2 \\ 0 & 1 & 0 & 0 \\ s_2 & 0 & c_2 & \ell_2 s_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$[T_3] = \begin{bmatrix} c_3 & 0 & s_3 & \ell_3 c_3 \\ 0 & 1 & 0 & 0 \\ -s_3 & 0 & c_3 & -\ell_3 s_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad [T_4] = \begin{bmatrix} c_4 & 0 & s_4 & \ell_4 c_4 \\ 0 & 1 & 0 & 0 \\ -s_4 & 0 & c_4 & -\ell_4 s_4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $c_i = \cos\theta_i$ and $s_i = \sin\theta_i$, $i = 1,4$.

Therefore, the frame x_4-y_4-z_4 with respect to x_o-y_o-z would be given by,

$$[T_1][T_2][T_3][T_4] = \begin{bmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & 1 & \ell_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_2 & 0 & -s_2 & \ell_2 c_2 \\ 0 & 1 & 0 & 0 \\ s_2 & 0 & c_2 & \ell_2 s_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times$$

$$\times \begin{bmatrix} c_3 & 0 & s_3 & \ell_3 c_3 \\ 0 & 1 & 0 & 0 \\ -s_3 & 0 & c_3 & -\ell_3 s_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_4 & 0 & s_4 & \ell_4 c_4 \\ 0 & 1 & 0 & 0 \\ -s_4 & 0 & c_4 & -\ell_4 s_4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & 1 & \ell_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_2 & 0 & -s_2 & \ell_2 c_2 \\ 0 & 1 & 0 & 0 \\ s_2 & 0 & c_2 & \ell_2 s_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_{3+4} & 0 & s_{3+4} & \ell_4 c_{3+4} + \ell_3 c_3 \\ 0 & 1 & 0 & 0 \\ -s_{3+4} & 0 & c_{3+4} & -\ell_4 s_{3+4} - \ell_3 s_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & 1 & \ell_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_{3+4-2} & 0 & s_{3+4-2} & \ell_4 c_{3+4-2} + \ell_3 c_{3-2} + \ell_2 c_2 \\ 0 & 1 & 0 & 0 \\ -s_{3+4-2} & 0 & c_{3+4-2} & -\ell_4 s_{3+4-2} - \ell_3 s_{3-2} + \ell_2 s_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} c_1 c_{3+4-2} & -s_1 & c_1 s_{3+4-2} & c_1(\ell_4 c_{3+4-2} + \ell_3 c_{3-2} + \ell_2 c_2) \\ s_1 c_{3+4-2} & c_1 & s_1 s_{3+4-2} & s_1(\ell_4 c_{3+4-2} + \ell_3 c_{3-2} + \ell_2 c_2) \\ -s_{3+4-2} & 0 & c_{3+4-2} & -\ell_4 s_{3+4-2} - \ell_3 s_{3-2} + \ell_2 s_2 + \ell_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

Since the object (fixed to the end effector) is to remain on the x_o-y_o plane while being moved, the origin of the x_4-y_4-z_4 frame has to lie on a plane parallel to the x_o - y_o plane, at a height h_o above it, assuming the thickness of the object is $2h_o$. From eqn. (2.1), we have

$$-\ell_4 s_{3+4-2} - \ell_3 s_{3-2} + \ell_2 s_2 + \ell_1 = h_o \tag{2.2a}$$

and also

$$c_1(\ell_4 c_{3+4-2} + \ell_3 c_{3-2} + \ell_2 c_2) = x \tag{2.2b}$$

$$s_1(\ell_4 c_{3+4-2} + \ell_3 c_{3-2} + \ell_2 c_2) = y \tag{2.2c}$$

where x and y are the specified x and coordinates of the object on the x_o-y_o plane. Now an additional condition, that the link 4 should be normal to the x_o-y_o plane, is imposed to avoid tilting of the object. From Fig. 2.6 we have,

$$\theta_2 - (\theta_3 + \theta_4) = -\pi/2$$

i.e. $$\theta_3 + \theta_4 - \theta_2 = \pi/2 \qquad (2.3)$$

therefore, $$\cos(\theta_3 + \theta_4 - \theta_2) = c_{3+4-2} = 0$$

and $$\sin(\theta_3 + \theta_4 - \theta_2) = s_{3+4-2} = 1$$

Substituting these into eqns. (2.2a), (2.2b) and (2.2c), we have,

$$-\ell_3 s_{3-2} + \ell_2 s_2 = \ell_4 - \ell_1 + h_o \qquad (2.4a)$$

$$c_1(\ell_3 c_{3-2} + \ell_2 c_2) = x \qquad (2.4b)$$

$$s_1(\ell_3 c_{3-2} + \ell_2 c_2) = y \qquad (2.4c)$$

From eqns. (2.4b) and (2.4c) we get,

$$\ell_3 c_{3-2} + \ell_2 c_2 = r, \quad \text{where} \quad r = \sqrt{x^2+y^2}$$

From this and eqn. (2.4a), we get,

$$\ell_2^2 + \ell_3^2 + 2\ell_2\ell_3\, c_3 = r^2 + (\ell_4 - \ell_1 + h_o)^2$$

or, $$\theta_3 = \cos^{-1}\left(\frac{r^2 + (\ell_4 - \ell_1 + h_o)^2 - \ell_2^2 - \ell_3^2}{2\ell_2\ell_3} \right)$$

Substituting this into (2.4a), we get θ_2. The expression for θ_1 can also be obtained from equations (2.4b) and (2.4c) as $\theta_1 = \mathrm{Tan}^{-1}(y/x)$. Now that θ_2 and θ_3 are known, θ_4 can be got

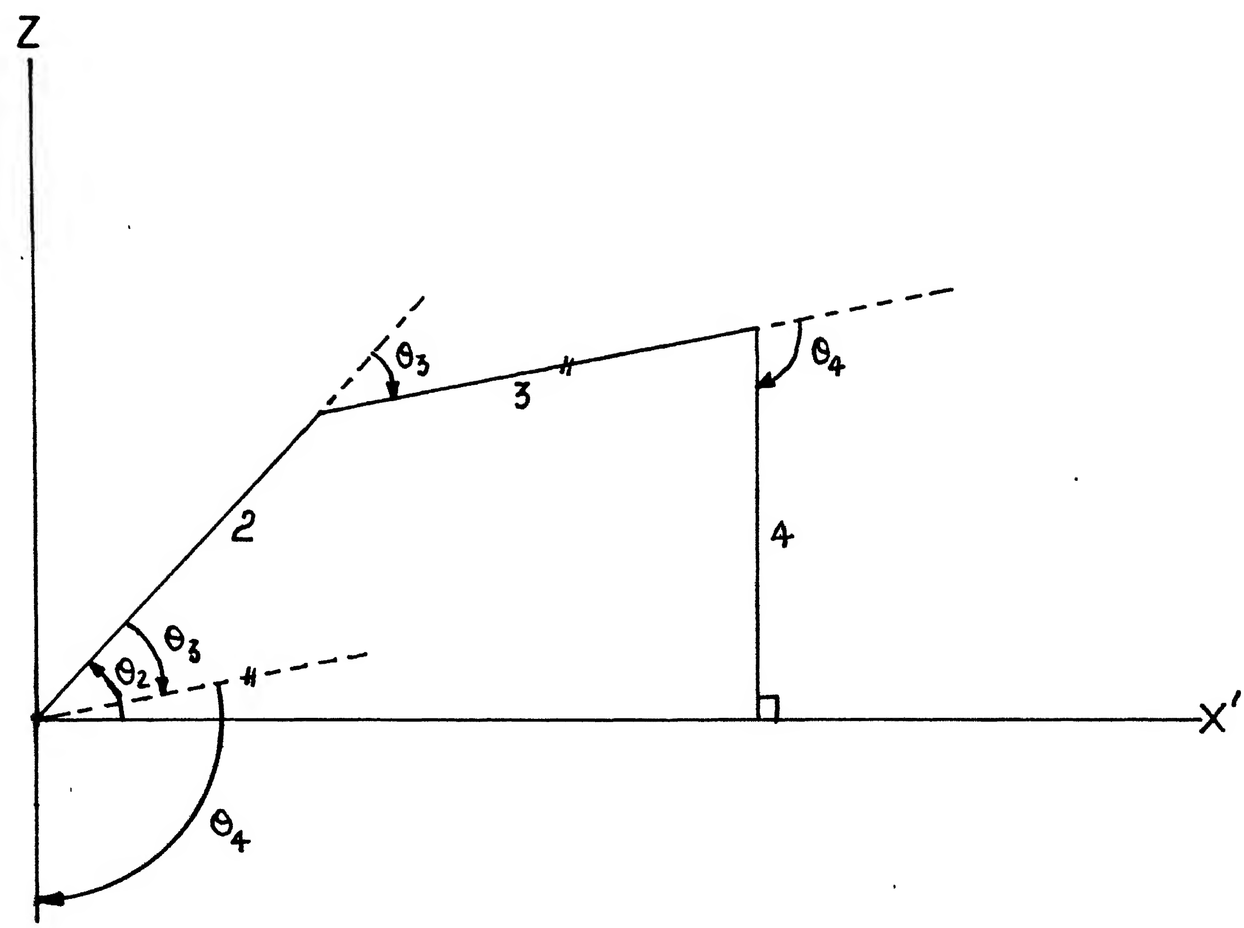


Fig. 2.6 Interdependence of the Joint Angles θ_2, θ_3 and θ_4

easily from the relation $\theta_3 + \theta_4 - \theta_2 = \pi/2$. So given the x-y coordinates of the object, all the joint angles θ_i, $i = 1,4$ can be computed using the above procedure. If the object being moved is a polygon, where rotational manoeuvring is necessary, the rotation of the end effector, θ_5, can be easily obtained as $\theta_5 = \phi + \theta_o$, where ϕ is the orientation of the object and θ_o is some constant. This is because, while the change in orientation of the object will be the same as the change in θ_5, the starting values of θ_5 and ϕ need not be the same.

With these, the joint angles are calculated for the starting and the goal positions, and the joint vectors $\vec{J}C = \{\theta_1\ \theta_2\ \theta_3\ \theta_4\ \theta_5\}^T$ [or $\{\theta_1\ \theta_2\ \theta_3\ \theta_4\}^T$ if there is no rotation] are formed. Thus the algorithm starts with the calculated $\vec{J}C_{start}$ and $\vec{J}C_{final}$ and computes the successive intermediate positions in the joint space as $\vec{J}C$ till $\vec{J}C = \vec{J}C_{final}$. The corresponding cartesian points can be easily computed by direct kinematics.

Now out of the five joint angles, θ_1 and θ_5 are independent of the rest. The values of θ_2, θ_3 and θ_4 are governed by two eqns. (2.3 and 2.4a). So only one of these three can be treated as an independent variable. Thus in the joint vector, only two of the components (or three if rotation of the object is also taken into account) are independent. The state of the robot can therefore be fully defined by two-dimensional (or three) vector and thesearch for the path can be carried out with this.

The other joint angles at any point on the path can be computed easily using eqns. (2.3) and (2.4a). From eqn. (2.3) we have,

$$\theta_3 - \theta_2 = \pi/2 - \theta_4$$

i.e. $$\sin(\theta_3 - \theta_2) = \cos\theta_4.$$

Substituting this in eqn. (2.4a), we get,

$$-\ell_3 \cos\theta_4 + \ell_2 \sin\theta_2 = \ell_4 - \ell_1 + h_o$$

If we treat θ_2 as an independent variable, we get

$$\theta_4 = \cos^{-1}\left(\frac{\ell_2 \sin\theta_2 - \ell_4 + \ell_1 - h_o}{\ell_3}\right) \qquad (2.5)$$

and from eqn. (2.3),

$$\theta_3 = \frac{\pi}{2} + \theta_2 - \theta_4 \qquad (2.6)$$

Let the reduced joint vector, we use for searching the path, be denoted by ${}_r\vec{JC}$, with the corresponding reduced vectors for the starting and final positions being ${}_r\vec{JC}_{st}$ and ${}_r\vec{JC}_f$ respectively.

The penalty function approach is based on an objective function which is to be defined, over the range of the variables involved. This objective function, when minimized using some minimization algorithm should lead to the destination which the object is supposed to reach. An objective function of the form $D_s(1 + \alpha_1 P_c + \alpha_2 P_\ell)$ is chosen for this purpose. D_s is the norm of the 'distance' separating the object in its present position from its destination. The word 'distance' has been used in a more general sense here, in that it accounts for the

difference in orientations as well, in case the motion involves rotational manoeuvring. The term D_s acts as a kind of pull towards the destination. The terms P_c and P_ℓ are the penalty terms which prevent the object from nearing an obstacle and the joint angles from going beyond the limits respectively. α_1 and α_2 are constants (penalty parameters for P_c and P_ℓ, respectively). The procedure for evaluation of each one of these terms is described below.

(i) Evaluation of D_s:

Let the reduced joint vector corresponding to some position of the object be ${}_r\vec{JC}$. D_s is now given by

$$D_s = ||{}_r\vec{JC}_f - {}_r\vec{JC}||$$

i.e.

$$D_s = [\sum_{i=1}^{2(3)} ({}_rJC_{fi} - {}_rJC_i)^2]^{1/2}$$

where ${}_rJC_{fi}$ and ${}_rJC_i$ are the i-th components of ${}_r\vec{JC}_f$ and ${}_r\vec{JC}$ respectively.

(ii) Evaluation of P_c:

The presence of this term, prevents the object from approaching any obstacle, too closely. Hence, the behaviour of this term should be such that it increases as the object nears the obstacle and the increase should be sharp when the object is close to the obstacle. A simple function which would behave like this is $1/d_{min}$ where d_{min} is the smallest distance which separates the object and the obstacle. This d_{min} is computed, as described in the algorithms discussed earlier in this section.

Since d_{min} can be calculated only in the cartesian frame, we will need the position of the object in the x_o-y_o frame (see Fig. 2.5). This can be done directly, using eqns. (2.5), (2.6), (2.4b) and (2.4c), once the reduced joint vector is known. The reduced joint vector would be available at any point along the path because the path will be generated directly in terms of the reduced joint vector.

The resultant penalty function, for all the obstacles put together can be defined as,

$$P_c = \sum_{i=1}^{ONUM} \frac{1}{d^i_{min}}$$

where, ONUM is the number of obstacles and d^i_{min} is the d_{min} between theo object and the i-th obstacle.

Here some saving in computation can be achieved by restricting the evaluation of the penalties for nearing the obstacles, to a subset of the complete set of obstacles. For any position of the object, the obstacles which have a significant effect on the subsequent motion of the object are only the ones which are sufficiently close to the present position of the object. In other words, only those obstacles 'i' for which $d^i_{min} < \delta$, where 'δ' is some small quantity, need be taken into consideration. So, a set of obstacles is dynamically maintained with obstacles being deleted from or appended to the set after every step which the object takes, according to the condition $d^i_{min} < \delta$. The penalties are in turn evaluated only with respect

to those obstacles which belong to this set and the summation $\Sigma\, 1/d^i_{min}$ is over the elements of this set.

(iii) Evaluation of P_ℓ:

The rotation of most joints in practice, will necessarily have limits, beyond which the corresponding joint angles cannot go. So any path which, while being traced, causes the joint angles to cross these limits will be useless. The term P_ℓ, keeps joint angles within the prescribed limits by imposing a penalty for approaching the joint angle limits.

Let θ^i_{min} and θ^i_{max} be the lower and upper limits of the i-th joint angle. So if the i-th joint angle has a value θ^i at some point along the path, we need to make sure that,

$$\theta^i_{min} \leq \theta^i \leq \theta^i_{max}$$

or $$0 < \hat{\theta}_i < 1, \quad \text{where} \quad \hat{\theta}_i = (\theta^i - \theta^i_{min})/(\theta^i_{max} - \theta^i_{min})$$

A term of the form $1/\hat{\theta}_i + 1/(1-\hat{\theta}_i)$ would keep $\hat{\theta}_i$ between 0 and 1 as required. So for all the joint angles together, we have,

$$P_\ell = \sum_{i=1}^{n_f} \left(\frac{1}{\hat{\theta}_i} + \frac{1}{1-\hat{\theta}_i}\right)$$

where n_f is the number of degrees of freedom of the robot being used.

Here too, given the reduced joint vector, the rest of the joint angles and in turn P_ℓ can be evaluated.

Given the reduced joint vector at any point along the path, the value of the objective function $D_5(1 + \alpha_1 P_c + \alpha_2 P_\ell)$ can now be evaluated, using the methods outlined above. So the objective function has effectively been defined as a function of the reduced joint vector. From what has been said above, about the evaluation of the terms D_5, P_c and P_ℓ, it is clear that all the three terms are everywhere positive. So if $\alpha_1 > 0$ and $\alpha_2 > 0$, the value of the objective function will always be greater than or equal to zero. The only point where the function value becomes zero is the destination since at this point $D_5 = 0$. Hence the destination, is where the global minimum of the objective function lies. It follows that this function, when minimized in an unconstrained fashion would lead to the destination (provided there are no local minima on the way). This minimization can be done using any of the known optimization algorithms. The Davidon-Fletcher-Powell algorithm is the best known method that uses gradients, so initially that was decided upon for this task. After testing it on some problems, it was found that since a path through a maze of obstacles involves frequent sharp turns, the DFP takes too many cycles of computation at each turn, though it eventually does converge to the minimum. Moreover since it is based on one-dimensional minimization along every new direction of search chosen, the path obtained is nowhere near the 'best' path to the destination. These factors prompted the use of a rather crude method in which

the gradient of the objective function is evaluated and then a small step is taken along the negative of the gradient repeatedly till the minimum is reached. In this case, it did turn out to be more efficient than using DFP, because of the consequent reduction in the path length drastically and also the avoidance of a one-dimensional minimization at every stage.

Problems involving rotational manoeuvring pose an additional problem, during minimization of the objective function. This is due to the presence of very small local minima with respect to the in plane rotation of the polygonal object.

Referring to Fig. 2.7, let O be the object with R being its reference point and let O_1 be the obstacle with R_1 its reference point. Let the line through R, perpendicular to $\overline{UW}$ of O_1 intersect the object and the obstacle at V and V_1 respectively. For simplicity let us assume that O and O_1 are rectangles with the side UW of O being parallel to side U_1W_1 of O_1. It is easily seen from Fig. 2.7 that d_{min} for this case is $|\overline{VV_1}|$, which is nothing but $|\overline{RV_1}| - |\overline{RV}|$. Now, if the orientation of the object is such that RW coincides with RV_1, we have $d_{min} = |\overline{RV_1}| - |\overline{RW}|$. Similarly if RU coincides with RV_1, $d_{min} = |\overline{RV_1}| - |\overline{RU}|$. Since $|\overline{RV}| < |\overline{RW}| = |\overline{RU}|$, we have $|\overline{RV_1}| - |\overline{RV}| > |\overline{RV_1}| - |\overline{RW}| = |\overline{RV_1}| - |\overline{RU}|$, or,

$$d_{min} > d_{min} = d_{min},$$

i.e. $1/d_{min} < 1/d_{min} = 1/d_{min}$

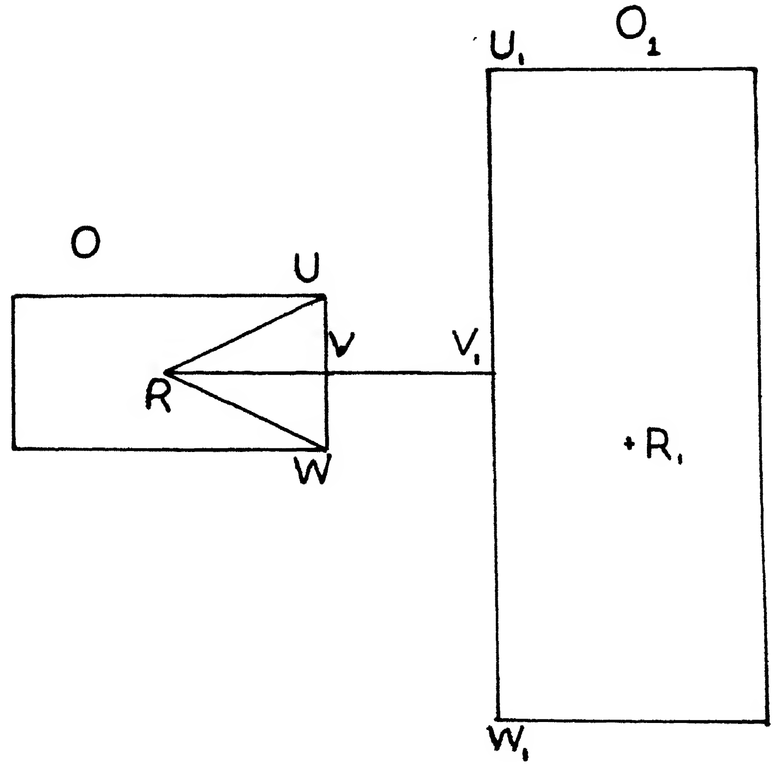


Fig. 2.7 Occurance of local minima with respect to Object Orientation

This shows that for the position shown in Fig. 2.7, the penalty for nearing the obstacle O_1 is in fact a local minimum with respect to the orientation, because the penalty increases along both the directions of rotation (U towards V and W towards V). This will prevent the object from rotating in any direction. This problem is eliminated by adding some hypothetical inscribing circles to the object. If the object is too elongated two inscribing circles at either end in the elongated direction are added. Otherwise, only one circle, the centre of which coincides with the reference point of the object is added. An illustration of this is given in Fig. 2.8.

In each case shown in Fig. 2.8, O is the object, R is its reference point and the O_{a}^{i}'s are the inscribing circles that have been added. The penalty for nearing an obstacle is now calculated as,

$$(1/d^i_{min}) = (1/d^i_{min,p}) + \left(\sum_{j=1}^{INC} 1/d^i_{min,j} \right)$$

where d^i_{min} is the resultant 'nearness' to the obstacle 'i', $d^i_{min,p}$ is the d_{min} between the polygonal object and the obstacle, and $d^i_{min,j}$ is the d_{min} between the j-th inscribed circle and the polygonal object, INC being the number of these inscribed circles. This additional summation term in the penalty function, has the effect of 'smoothening' out the small local minima which occur with respect to the inplane rotation of the object, as explained above.

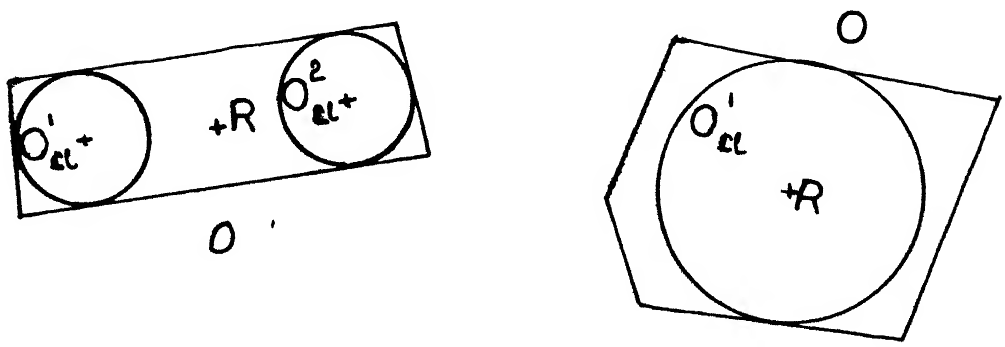


Fig. 2.8 Hypothetical Inscribing Circles being added to 'Smoothen out' the Objective Function

Minimization based on gradients is allowable only if the objective function is differentiable everywhere in the range of the variables involved. So the differentiability of the function we have chosen, must be proved before any of these minimization algorithms based on gradients, can be used. The objective function we have chosen is of the form $D_s(1+ \alpha_1 P_c + \alpha_2 P_\ell)$. D_s being given by $[\sum_{i=1}^{2(3)} ({}_r{}^{JC}{}_{fi} - {}_r{}^{JC}{}_i)^2]^{1/2}$ is obviously differentiable with respect to ${}_r{}^{JC}{}_i$ for all ${}_r i$. P_ℓ is given by,

$$\sum_{i=1}^{n} \left(\frac{1}{\hat{\theta}_i} + \frac{1}{1 - \hat{\theta}_i}\right)$$

where,

$$\hat{\theta}_i = \frac{\theta^i - \theta^i_{min}}{\theta^i_{max} - \theta^i_{min}} .$$

This too is differentiable with respect to θ^i every where except $\theta^i = \theta^i_{max}$ and $\theta^i = \theta^i_{min}$. Since the whole idea of having a term of this sort is to prevent the joint angles from going beyond the limits, the joint angles would be confined to the range,

$$\theta^i_{min} < \theta^i < \theta^i_{max} ,\quad \text{where it is differentiable.}$$

Proving the differentiability of P_c is not as straight forward as it was for D_s and P_ℓ because we do not have any closed form

expression for this. The form we have chosen for P_c is $1/d_{min}$ where d_{min} is as defined at the beginning of this section. Differentiability of a function of this form has been proved by Elmer G. Gilbert and Daniel W. Johnson [8]. Now, since all the three terms D_s, P_c and P_ℓ are differentiable, the complete objective function $D_s(1 + P_c\,\alpha_1 + P\;\;\alpha_2)$ is also differentiable over the range of the joint angles, which constitute the variables for the present problem.

2.3 The Configuration Space Approach

This approach is based on building a "configuration space" by expanding the obstacles, which will in turn enable us to treat the object to be moved as a point. Using the same representation for objects and obstacles as described in Fig. 2.3 , the object is shrunk to a point at its reference point and the obstacles are expanded accordingly. Now if the reference point of the object is always confined to the free space, i.e. the complement of the space occupied by the obstacle, then collisions will not occur. So the problem of finding the path for an object from one position to another reduces to that of finding one for a point through the free space in the presence of the expanded obstacles. Specific algorithms for finding the path using this approach are presented below.

2.3.1 Circular Object

Restricting ourselves to two dimensions, let us first consider the problem of moving a circular object which does not require any rotational manoeuvring through a maze of obstacles, all of which are polygonal.

The first step is to expand the obstacle so as to allow the object to be shrunk to a point (in this case, the centre of the circular object). The algorithm for this is quite straight forward.

Referring to Fig. 2.9, let V_1 be a vertex of the obstacle, the adjacent vertices being V_2 and V_3. Let the object be of radius r. Assuming a margin of safety 'm' is to be provided, let $d = m+r$. Now if the object is shrunk to a point at its centre, then the distance between the point and any of the edges of the obstacle cannot be less than 'd', if a collision is to be avoided (within the limits of safety 'm'). So if the object is reduced to a point, each edge of the obstacle will have to be moved out by a distance 'd'. Let the two edges originally incident on V_1 intersect at a new point V after being moved outward and let a and b be the feet of the perpendiculars dropped onto the new positions of these edges. It is easy to show that $L = d/\sin(\eta/2)$ where η is the internal angle at V_1. η can be computed easily from the coordinates of V_1, V_2 and V_3, using the cosine rule as

$$\eta = \cos^{-1}\left(\frac{s_{12}^2 + s_{13}^2 - s_{23}^2}{2s_{12}\, s_{13}} \right) \tag{2.7}$$

where, $s_{ij} = [(x_i-x_j)^2 + (y_i-y_j)^2]^{1/2}$ is the distance between the vertices V_i and V_j. It is clear from Fig. 2.9 that the line VV_1 bisects angle $V_2V_1V_3$. From the coordinates of V_1, V_2 and V_3 the slope of this line and hence the angle it makes with

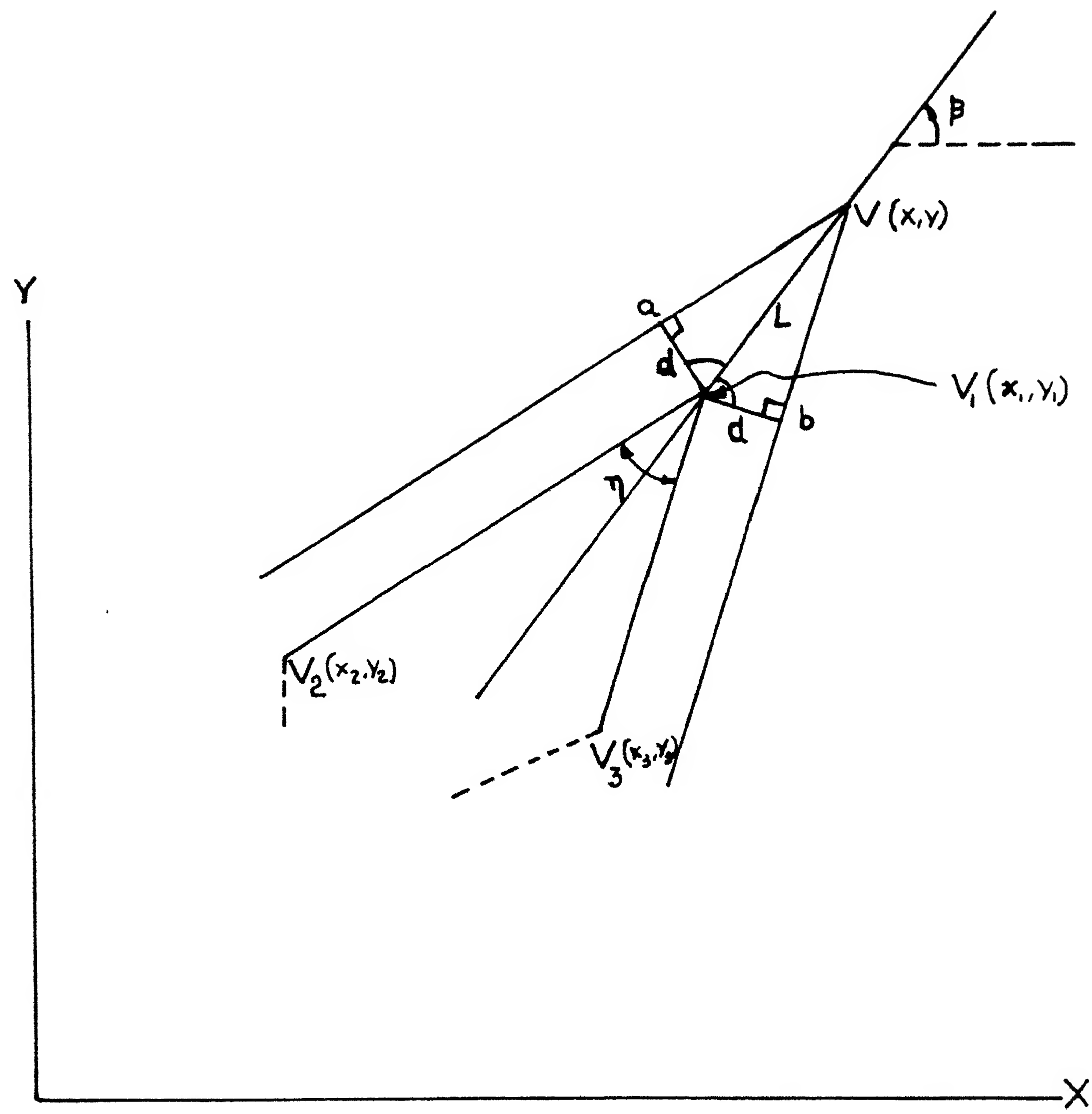


Fig. 2.9 Expansion of a Polygonal Obstacle by 'd' on every side

the x-axis (global) can be easily determined. Let this angle be β. So, if (x, y) and (x_1, y_1) are the coordinates of V and V_1 respectively, we have,

$$x = x_1 + L \cos \beta \qquad (2.8a)$$

and

$$y = y_1 + L \sin \beta \qquad (2.8b)$$

Therefore, given the coordinates of the vertices of the obstacle in the global frame, the coordinates of the vertices of the new expanded obstacle can be obtained from eqns. (2.8a) and (2.8b).

The above algorithm is of complexity $O(n)$ where 'n' is the number of vertices of the obstacle, since it is evident from the procedure given that the algorithm needs just one pass around the list of vertices of the obstacle.

Having expanded the obstacles, we now have to look for a path for a point through the free space. The algorithm for this is presented below.

As shown in Fig.(2.10), let A and B be the starting and goal points respectively. Let O_1, O_2 and O_3 be the expanded obstacles. Now the simplest path from A to B is a straight line joining the two, provided it is feasible. So, a straight line joining A and B is first drawn and is checked for intersections with the obstacles. If there are no such intersections, then a path has been found. Otherwise, let O_1 be the obstacle which comes first on the way from A to B along the straight line. Let the two points of intersection of $\overline{AB}$ with O_1 be a and c. Now a line (pq) perpendicular to $\overline{AB}$ is drawn at b,

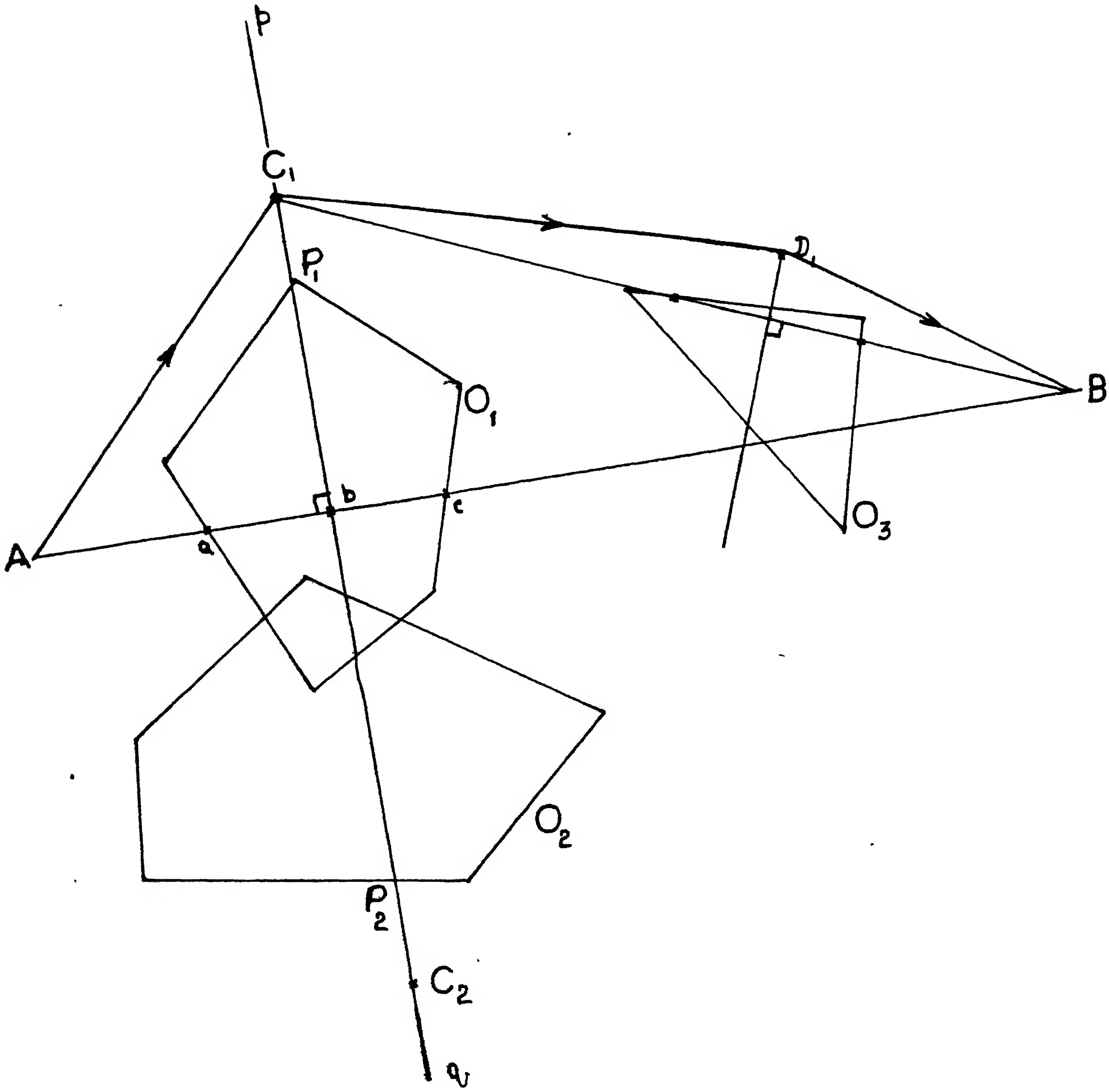


Fig. 2.10 Search for a Path between Points A and B

which is the mid point of $\overline{ac}$. This line is searched for points nearest to $\overset{c,\!?}{b}$ on either side of the line AB and not lying within any of the obstacles. These two points have been marked as P_1 and P_2 in Fig. 2.10. Each one of these two points would obviously be on the edge of some obstacle. So if we look for a path from A to B, passing through one of these two points P_1 and P_2, we would be unnecessarily forcing the path to graze the boundaries of the obstacles. The points P_1 and P_2 are therefore offset from the obstacles. Let the new points obtained be C_1 and C_2. How the points P_1 and P_2 are searched for and later offset to C_1 and C_2 respectively, has been explained a little later. Having obtained, C_1 and C_2, one of these two, say C_1 is chosen, based on a criterion which has been explained in detail later. Now, the original problem of finding a path from A to B is decomposed into two subproblems - (i) that of finding a path from A to C_1 and (ii) that of one from C_1 to B, solutions of which when put together yield a solution to the original problem. These two subproblems are now solved separately, again using the same procedure as was used for the original problem. The original straight line joining A and B is recursively modified to ultimately yield a path composed of straight line segments, not intersecting any of the obstacles.

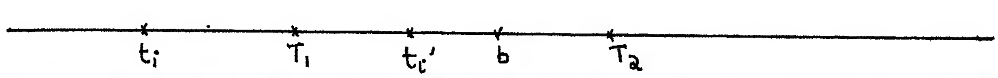
Two of the steps in the algorithm, namely (i) that of searching through the line pq (Fig. 2.10) for the points P_1, P_2 and then the points C_1 and C_2. (ii) that of choosing between

C_1 and C_2, for decomposing the path, have been described below in detail.

(i) In Fig. 2.11, let A, B, a, b, c and $\overline{pq}$ be what they were in Fig. 2.10. To locate the points P_1 and P_2 on pq, the points of intersection of pq with each of the obstacles are first determined. Let the distances of the corresponding points of obstacle O_i from b be t_i and t_i where $t_i \leq t_i'$, such that the distance is negative if the point lies between p and b and positive, if it lies between b and q.

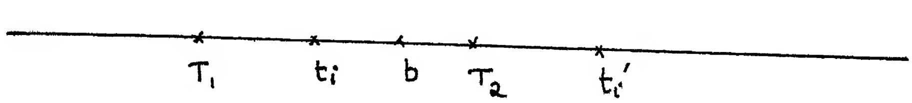
Let the point 'b' lies in some obstacle O_k (O_1 in Fig. 2.11) and let the corresponding points of intersection with pq be at distances t_k and t_k' from b. Let T_1 and T_2 be the required distances of P_1 and P_2 from b. We first set $T_1 = t_k$ and $T_2 = t_k'$. Now let t_i and t_i be the points of intersection of some obstacle O_i, other than O_k with pq. The values of T_1 and T_2 are now updated depending on the values of t_i and t_i'. Here there are several possibilities.

(a) $t_i < T_1$ and $T_1 < t_i' < T_2$:



In this case T_1 is updated to t_i by setting $T_1 = t_i$ and T_2 is left as it is.

(b) $T_1 < t_i < T_2$ and $t_i' > T_2$:



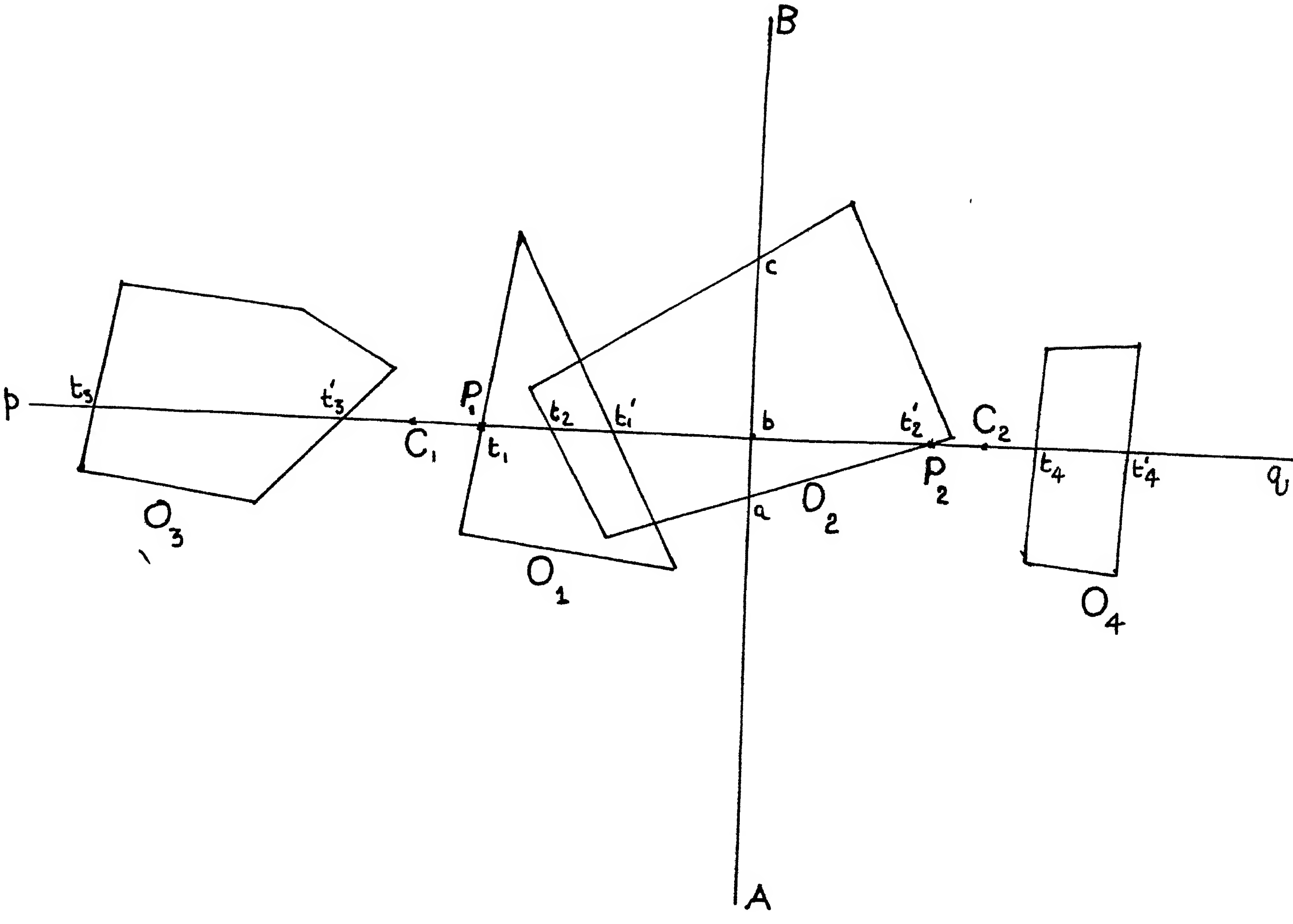


Fig. 2.11 Search for the Intermediate Points

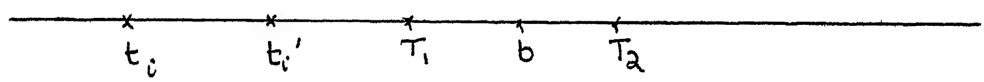
Here we set $T_2 = t_i'$ and T_1 is left as it is.

(c) $t_i < T_1, T_2 < t_i'$:

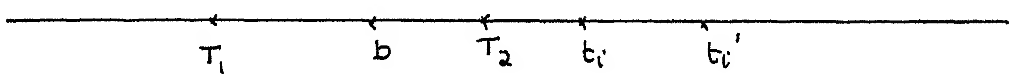
Here we set $T_1 = t_i$ and $T_2 = t_i'$.

For all other cases, T_1 and T_2 are left as they are. This is carried out for all i = 1, ONUM, $i \neq k$, where ONUM is the number of obstacles. The values of T_1 and T_2 obtained finally give the distances of P_1 and P_2 respectively from b.

Having obtained P_1 and P_2, they are now offset to C_1 and C_2 as follows. If, while comparing t_i, t_i' with T_1, T_2, a situation arises where $t_i' < T_1$, then



the gap $dt_i = T_1 - t_i'$ is recorded. Similarly, if $t_i > T_2$, then,



again the gap $dt_i' = t_i - T_2$ is recorded. Let Δt_1 and Δt_2 be the smallest of the dt_i' s and dt_i' 's, respectively. The distances of the points C_1 and C_2 from b are now found as $T_{c1} = T_1 - \Delta t_1/2$ and $T_{c2} = T_2 + \Delta t_2/2$ respectively. If, there is no obstacle for which, the first of the above two conditions is satisfied, then, some multiple of the margin is chosen arbitrarily as the offset, and T_{c1} becomes $T_1 - \mu m$ where 'μ' is an integer which has been chosen arbitrarily depending on the problem, and 'm' is the margin.

Similarly, if the second condition is not satisfied for any obstacle, then T_{c2} is found as $T_{c2} = T_2 + \mu . m$.

(ii) Having obtained the two intermediate points C_1 and C_2, for the path from A to B (Fig. 2.10), the next step is to make a choice, not necessarily final, between C_1 and C_2.

The procedure adopted for this is the A^* algorithm [9]. The digraph needed for the A^* algorithm to be employed is built as follows.

Points A and B alongwith every intermediate point obtained during the recursion process form the nodes of the graph, as shown in Figures 2.12 and 2.13. An edge exists between any two nodes, corresponding to points N_1, N_2, if and only if the path from N_1 to N_2 has come out to be the line segment $\overline{N_1N_2}$ at some stage in the recursion process. This will happen only when the line segment $\overline{N_1N_2}$ does not intersect any obstacle. The graph for the situation in Fig. 2.12 would be as in Fig. 2.13.

A^* is now used to look for the optimum path from A to B in this weighted digraph where the weight for an edge $\overline{N_1N_2}$ is the length of the segment joining the points corresponding to nodes N_1, N_2.

This graph can be pruned, before the search is carried out, so that unnecessary search for a path through points, for which it is known that a path is not feasible, can be avoided. The conditions under which a node will be deleted from the graph are as follows.

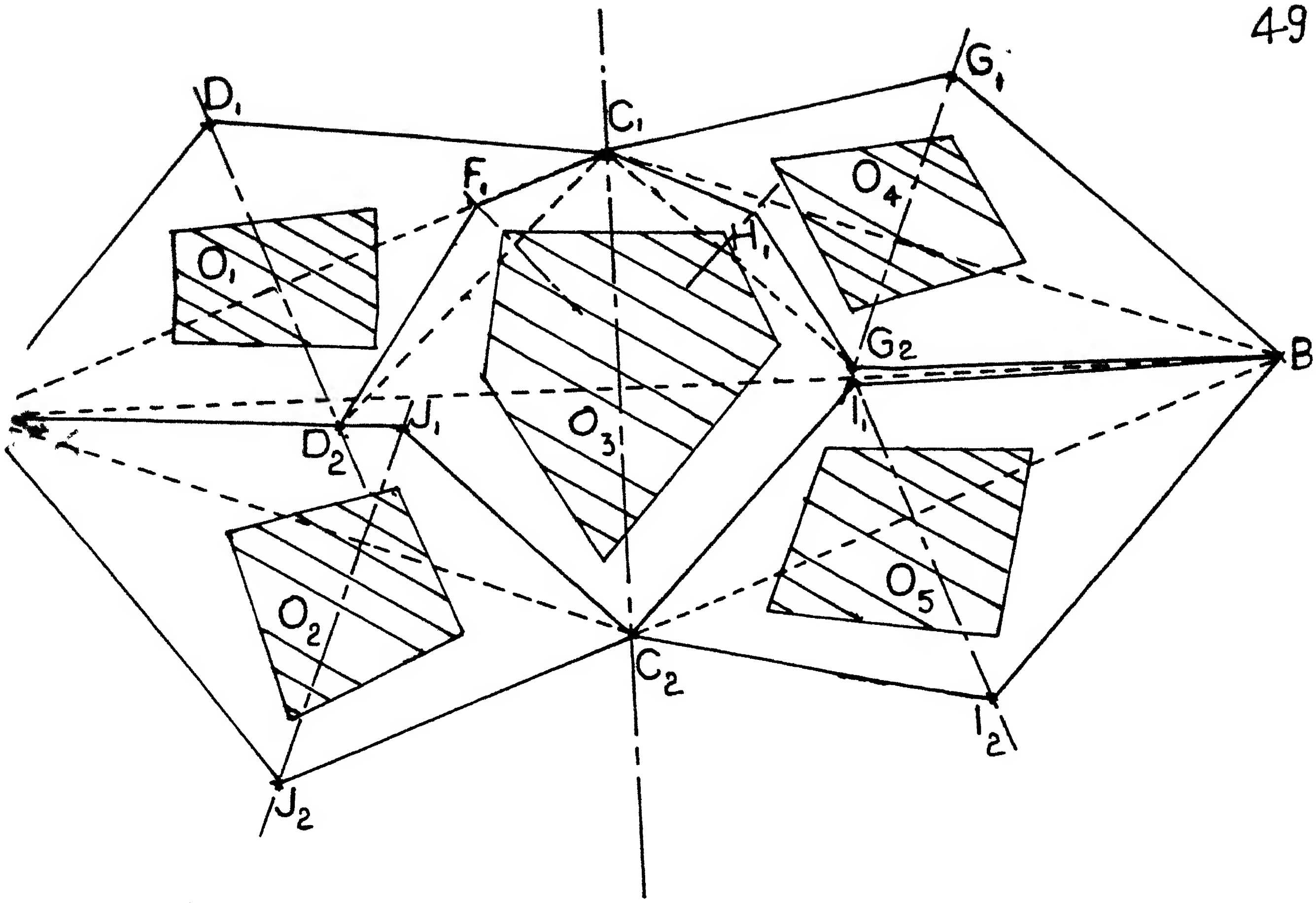


Fig. 2.12 Generation of all the Intermediate Points for the path from A to B

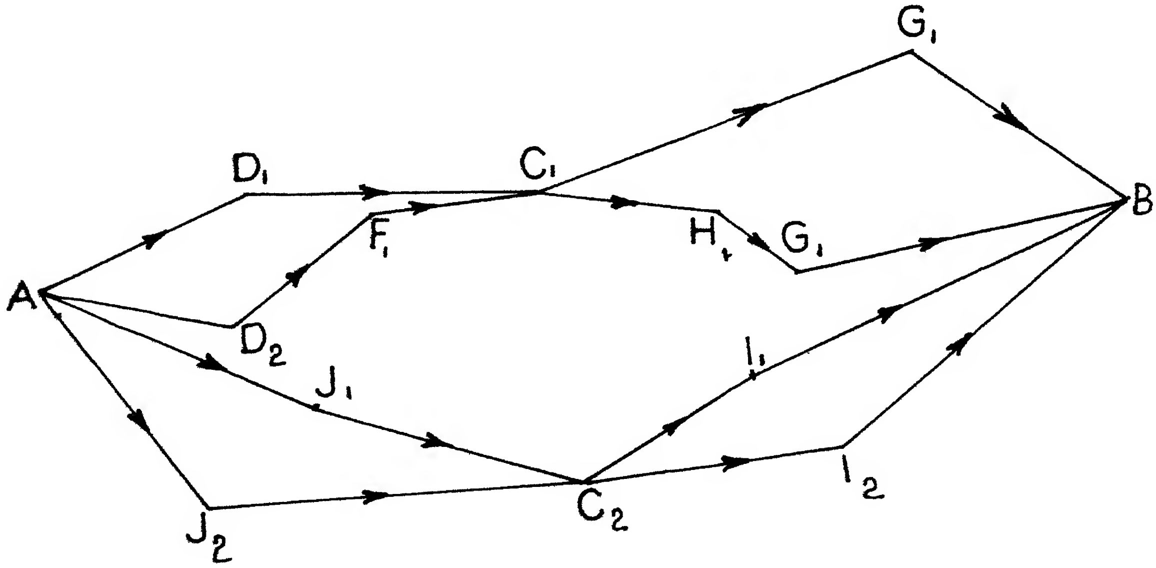


Fig. 2.13 Digraph for the situation in Fig. 2.12

(i) When the point corresponding to the node is beyond the limits of the workspace of the robot, the path through that point obviously becomes infeasible. Hence that point is rejected.

(ii) Referring to Fig. 2.14, let C_1 and C_2 be the intermediate points for the path from A to B. If C_1 is now chosen to lie along the path, for the path from A to C_1, we have two more intermediate points D_1 and D_2. It is easy to see from Fig. 2.14 that the intermediate point for the path from A to C_1 will have to lie on the same side of pq as A, if unnecessary repetitions are to be avoided. In Fig. 2.14, D_2 does not satisfy this condition and is hence rejected. A similar condition is applied to the intermediate points for the path from C_1 to B. In this case, the two will have to lie on the same side of line pq as B.

Likewise if C_1 and C_2 are the intermediate points for the path from A to B, C_1 and C_2 are first checked for the above two conditions. Three possibilities could arise here - (i) both C_1 and C_2 are infeasible points, (ii) only one of them is infeasible (iii) both are feasible.

Case (i) implies that, there does not exist a feasible path from A to B. Case (ii) implies that a feasible path from A to B will have to pass through the intermediate point which is feasible. In case (iii), the path from A to B could either pass through C_1 or through C_2. If C_1 is chosen to lie on the path, then we need to look for a path from A to B, passing through C_1 in the digraph for the problem.

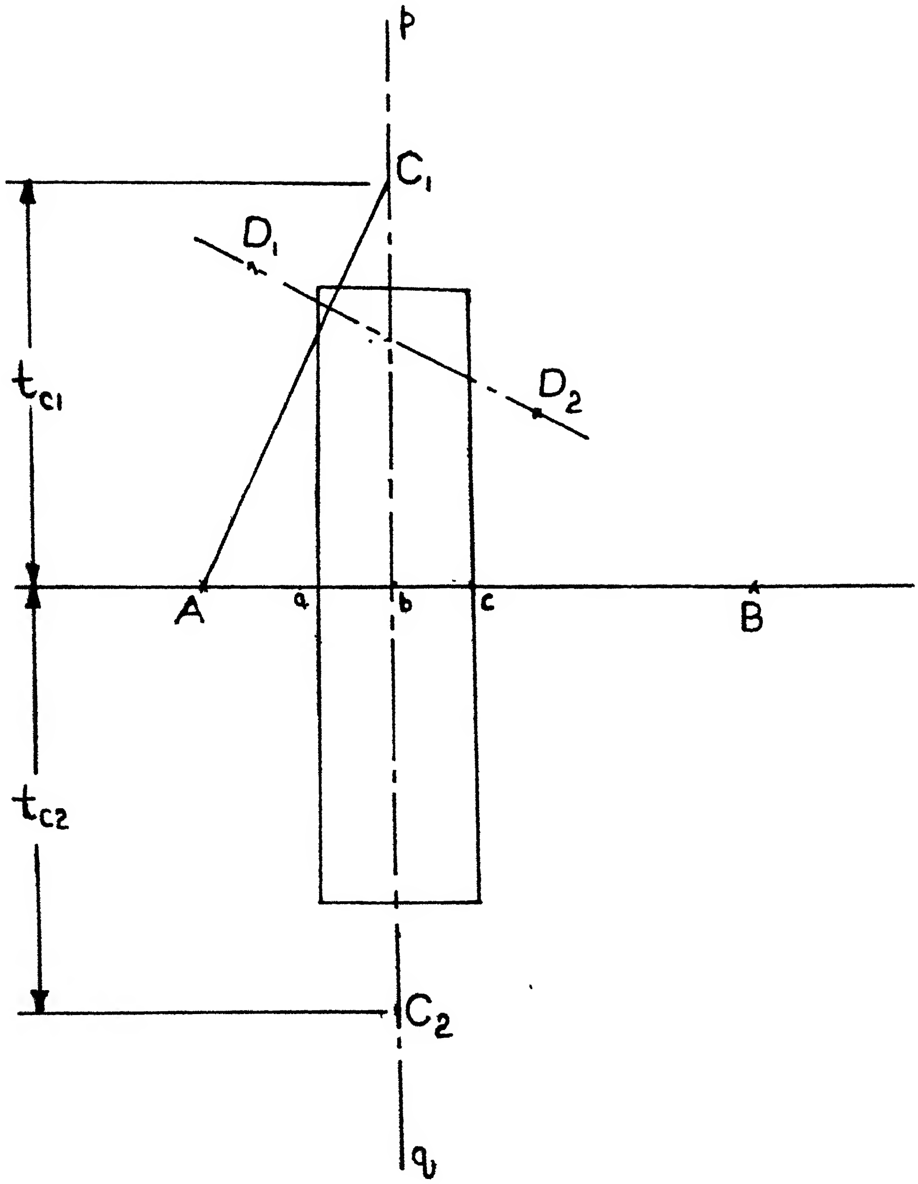


Fig. 2.14 Condition for eliminating a node

The algorithm A^* associates a heuristic function $h(C)$ with every node C of the graph. If we are looking for a path from A to B, then $h(C)$ is such that it is a lower bound on the actual path length from A to B through C. For a heuristic function $h(C)$ chosen in this fashion, A^* assures an optimal path from A to B if one exists. The lower bound on the path length between any two nodes is obviously given by the length of the straight line joining the points corresponding to the two nodes. Hence, the lower bound on the path from A to B, through C is $|\overline{AC}| + |\overline{CB}|$. From Fig. 2.14, we have,

$$|\overline{AC_2}| + |\overline{C_2B}| \begin{Bmatrix} > \\ < \\ = \end{Bmatrix} |\overline{AC_1}| + |\overline{C_1B}| \quad \text{iff} \quad t_{c2} \begin{Bmatrix} > \\ < \\ = \end{Bmatrix} t_{c1}$$

where T_{c2} and t_{c1} are the distances of C_1 & C_2 respectively from b. This suggests that t_{ci} can itself be used as the heuristic function associated with the node corresponding to the poiht C_i.

If C_1 and C_2 are the intermediate points obtained, and a choice is to be made between the two, then, the nodes are ordered first based on the values of $h(C_1)$ and $h(C_2)$, in the increasing order of the values of these functions. After this, the node with the lowest 'h' is chosen as the intermediate point. If the path through this turns out to be infeasible, then the next node in the order is tried out. If this also becomes infeasible, then it means, a path from A to B does not exist.

The algorithm, as given above, will definitely find a path if one exists, but the path obtained in some cases, may not be the

best possible path between the two end points. A typical situation where this could occur is shown in Fig. 2.15a.

In Fig. 2.15a, O is the expanded obstacle and A and B are the initial and final points respectively. The path obtained using the algorithm as given above is $AD_1C_1I_1H_1G_1F_1J_1B$. It is easily seen from Fig. 2.15a that $AH_1G_1F_1B$ is obviously a better path than what has been found. To avoid such inefficiencies in the path obtained, the part of the path generated at every stage of the recursion process is refined. The refinement is done as follows.

Let the path obtained between two points A_i and B_i, at some stage in the recursion be, as shown schematically in Fig. 2.15b.

The intermediate points are $C_iD_iJ_iF_iG_iH_iI_i$ generated in that order. Now paths between pairs of vertices are checked for intersections with the obstacles. These pairs are chosen in the order given below.

A_iI_i, A_iG_i, A_iH_i, J_iB_i, J_iI_i, J_iG_i, J_iH_i, D_iB_i,

D_iI_i, D_iG_i, D_iH_i, F_iB_i, F_iI_i, F_iG_i, F_iH_i.

Let the first pair of points between which a path, not intersecting any of the obstacles exists, be say J_iG_i. Then the path from J_i to B_i is modified to $J_iG_iI_iB_i$ and the nodes between J_i and G_i are deleted. If this does not happen for any of the pairs, then the path is left as it is.

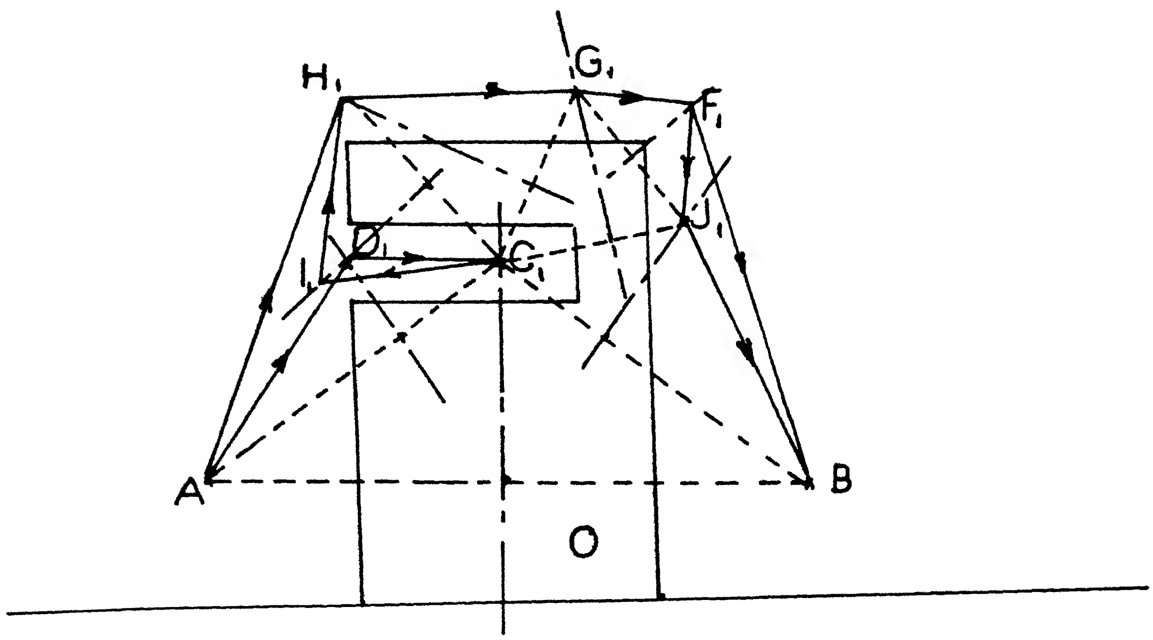


Fig. 2.15a A typical case of path inefficiency

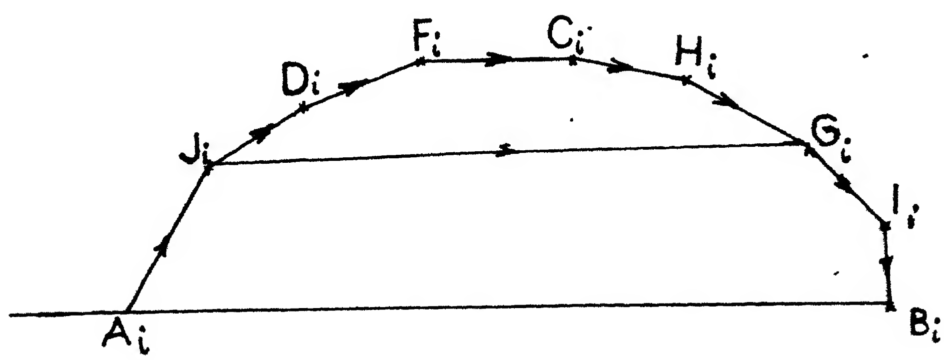


Fig. 2.15b Refinement of path

The path obtained at every stage of the recursion is refined in the manner outlined above. The final path obtained after this would be the best possible path between the two end points.

This in effect, solves the problem of finding the path for a circular object to be moved from one point to another, in a plane, through any arbitrary arrangement of polygonal obstacles.

2.3.2 Polygonal Object

Here we consider the problem of moving a polygonal object which has rotational freedom as well through a maze of polygonal obstacles from a specified starting position to a specified end position. Here also, we use the same representational scheme as described in Section 2.2 along with Fig. 2.3. The idea is essentially the same as the one outlined above for circular objects, except that the problem becomes a little more complicated owing to another degree of freedom i.e. rotation, for the object. Here too, the object is first reduced to a point, which coincides with its reference point and the obstacles are suitably expanded. The envelope that is obtained while expanding the obstacle, will obviously depend on the orientation of the object. Hence we end up with the grown obstacles being embedded in a 3-D space (extra dimension for the rotation). These grown obstacles are not polyhedra, but have curved surfaces, which makes the task of dealing with them even more tougher.

The object orientation (as defined in Fig. 2.3) may be subjected to a restriction that it can lie only between ϕ_{min} and ϕ_{max}. This range is divided into N equal parts and the envelopes for each of these N+1 discrete orientations of the object, are created around every obstacle. The allowable orientations are given by,

$$\phi_\lambda = \phi_{min} + \lambda \left(\frac{\phi_{max} - \phi_{min}}{N} \right), \quad \lambda = 0,1,\ldots N.$$

The algorithm for generating these envelopes is given below.

Referring to Fig. 2.16, let $V_1V_2V_3V_4$ be the obstacle and $v_1v_2v_3v_4$, the object at some fixed orientation. Let R and R_1 be the reference points of the obstacle and the object respectively. Now a vertex, say V_1, of the obstacle is chosen along with some direction of traversal. Let this direction be clockwise (indicated by an arrow in Fig. 2.16) with respect to R. The object is now placed such that the point R and the object are on opposite sides of the line through V_1 and V_2 without touching the obstacle, $\overline{V_1V_2}$ being the edge starting from V_1, along the direction of traversal.

Here, lines parallel to V_1V_2 are drawn through each on of the vertices of the object. Let the one closest to V_1V_2 be ℓ_1 with the corresponding vertex of the object being v_1. The envelope is now generated with V_1 and v_1 as the starting vertices.

Having determined the starting points V_1 and v_1 in the obstacle and the object respectively, V_1 and v_1 are made to

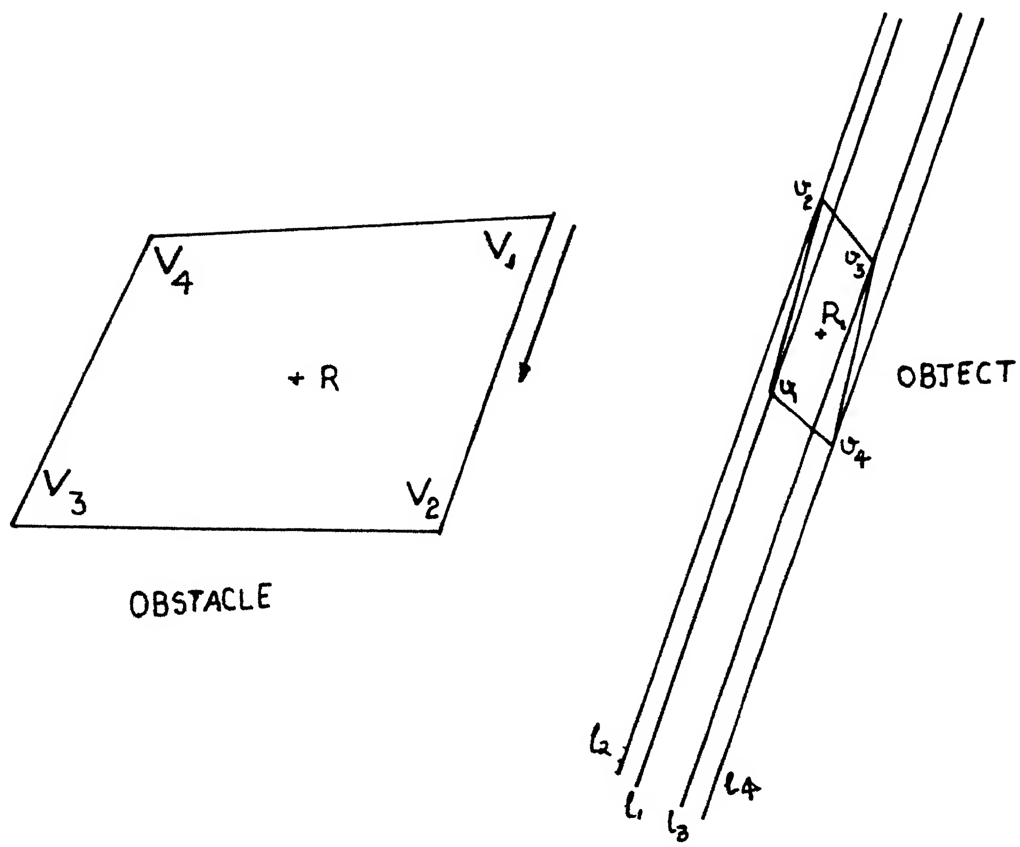


Fig 2·16 Choice of the starting pair of vertices for generating the envelope.

coincides, with the orientation remaining the same. The reference point R_1, of the object at this position becomes one of the vertices of the envelope. In Fig. 2.14, this vertex is V_{el}.

Referring to Fig. 2.17, let v_1v_2 be the clockwise edge (with respect to r) of the object from v_1. The angle $v_2v_1(V_1)V_2$ measured outside the obstacle, is now determined and depending on the value obtained, the next course of action is decided upon. Two different possibilities are discussed below.

(a) Angle $v_2v_1\ (V_1)V_2 < \pi$:

In this case, the object is translated along the direction parallel to V_1V_2 till v_1 coincides with the vertex V_2 of the obstacle.

(b) Angle $v_2v_1\ (V_1)V_2 > \pi$:

Here the object is translated in the direction parallel to v_1v_2 until v_2 coincides with V_1.

After one of these is done, the new position of the reference point of the object gives the next vertex of the envelope. This is carried out repeatedly till the object comes back to its starting position. The envelope, which is also a polygon, is obtained by joining the successive positions of the reference point of the object.

It is easily seen that the maximum number of sides of the envelope will be m+n where m and n are the number of sides of obstacle and the object respectively.

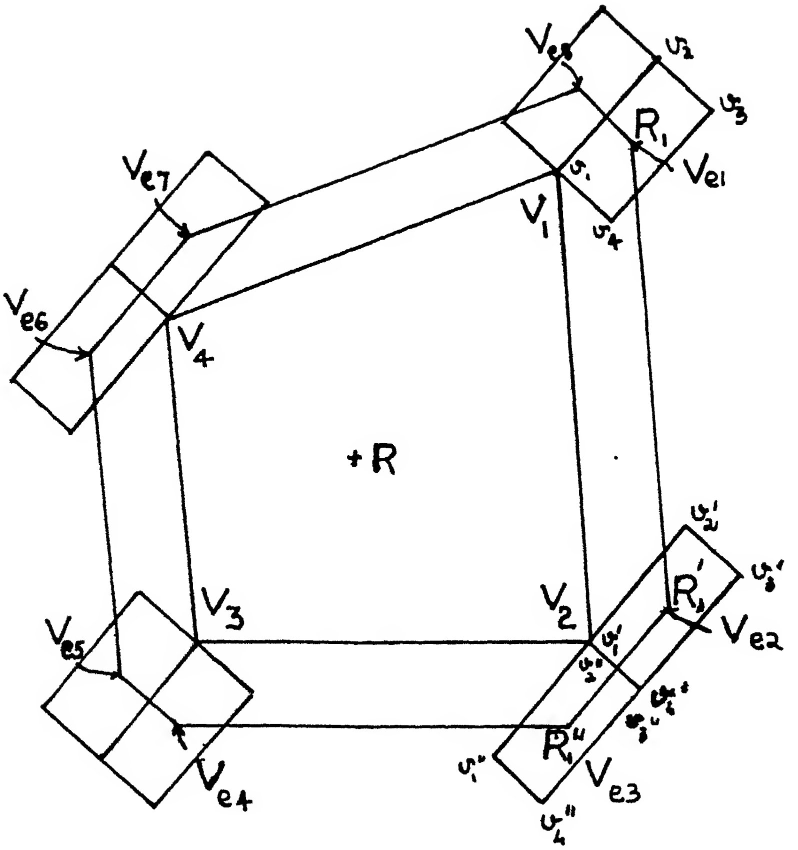


Fig. 2.17 Envelope Generation

Envelopes for each orientation ϕ_λ of the object, are represented by plates of thickness equal to the constant rotation interval chosen $(\phi_{max} - \phi_{min})/N$ and shaped like the envelopes at the corresponding ϕ_λ's. These plates are then stacked oneon top of the other, in the order ϕ_λ's, with the plate corresponding to ϕ_{min} at the bottom and that corresponding to ϕ_{max} at the top, to give approximately the expanded obstacle in 3-D. This is done in turn for all the obstacles in the workspace. These plates forming the equivalent 3-D obstacle are usually referred to as slices [4].

The problem of finding the path from one position to another, on a plane, for a polygonal object, such that it avoids all the polygonal obstacles on the way, now gets reduced to that of finding the path of the reference point of the object, from one point to another, in 3D-space through the maze of 3D obstacles.

Let the initial and final positions of the object be given by (x_i, y_i, ϕ_i) and (x_f, y_f, ϕ_f). As was done in Sec. 2.3.1, the search is started by first joining the initial and final points by a straight line and then checking for intersections with obstacles on the way. For any given obstacle, this is done as follows.

For any point (x, y, ϕ) on the line joining (x_i, y_i, ϕ_i) and (x_f, y_f, ϕ_f) we have,

$$\frac{x - x_i}{x_f - x_i} = \frac{y - y_i}{y_f - y_i} = \frac{\phi - \phi_i}{\phi_f - \phi_i}$$

or $$x = x_i + (x_f - x_i)\left(\frac{\phi - \phi_i}{\phi_f - \phi_i}\right) \qquad (2.9a)$$

and $$y = y_i + (y_f - y_i)\left(\frac{\phi - \phi_i}{\phi_f - \phi_i}\right) \qquad (2.9b)$$

ϕ is now varied from ϕ_i to ϕ_f in small steps. For each such ϕ, the corresponding x and y are calculated from eqns. (2.9a) and (2.9b). The slice nearest to this value of ϕ is given by an integer j, where j is the nearest integer to $[(\phi - \phi_{min})/\Delta\phi]$ with $\Delta\phi = (\phi_{max} - \phi_{min})/N$. A check is then made to see if the point (x, y) lies inside the j-th slice. The first value of ϕ (starting from ϕ_i) for which this happens gives the point where the line joining (x_i, y_i, ϕ_i) to (x_f, y_f, ϕ_f) 'enters' the obstacle. Similarly, the last value of ϕ for which this happens would be the point where the line 'exits' from the obstacle. Let these two values of ϕ be ϕ_a and ϕ_c with their corresponding x, y coordinates being (x_a, y_a) and (x_c, y_c) respectively. Then the points of intersection of the line with the obstacle are

$$(x_a, y_a, \phi_a) \text{ and } (x_c, y_c, \phi_c).$$

A check of this sort is made for intersections of the line with each one of the obstacles. If it does not intersect any of the obstacles, then a path has been found. Otherwise let the points of intersection with the obstacle which is encountered first (starting from (x_i, y_i, ϕ_i)), be (x_a, y_a, ϕ_a)

and (x_c, y_c, ϕ_c). Let (x_b, y_b, ϕ_b) be the midpoint of the segment joining these two points. Now a plane (P_n) at (x_b, y_b, ϕ_b) normal to the line joining (x_i, y_i, ϕ_i) and (x_f, y_f, ϕ_f) is drawn. The intermediate points are now located as follows.

Let the vector joining (x_i, y_i, ϕ_i) to (x_f, y_f, ϕ_f) be $\vec{if} = x_{if}\,\hat{i} + y_{if}\,\hat{j} + \phi_{if}\,\hat{k}$, where $x_{if} = x_f - x_i$, $y_{if} = y_f - y_i$ and $\phi_{if} = \phi_f - \phi_i$. Consider a unit vector lying on the plane P_n, parallel to the x - y plane at b (see Fig. 2.18a). This vector would be of the form $\bar{x}\,\hat{i} + \bar{y}\,\hat{j}$ where,

$$\bar{x} = \frac{-y_{if}}{\ell} \text{ and } \bar{y} = \frac{x_{if}}{\ell} \text{ where } \ell = [x_{if}^2 + y_{if}^2]^{1/2}$$

because $(\bar{x}\,\hat{i} + \bar{y}\,\hat{j}).(x_{if}\,\hat{i} + y_{if}\,\hat{j} + \phi_{if}\,\hat{k}) = 0$. Let these values of $\bar{x}$ and $\bar{y}$ be denoted by x_ϕ and y_ϕ respectively. Let the unit vector be called $\hat{\phi} = x_{\phi}\hat{i} + y_\phi\,\hat{j}$. Consider also the vector $\vec{V}$ normal to both $\hat{\phi}$ and $\vec{if}$, i.e.

$$\begin{aligned}
\vec{V} &= \hat{\phi} \times \vec{if} \\
&= (x_\phi\,\hat{i} + y_\phi\,\hat{j}) \times (x_{if}\,\hat{i} + y_{if}\,\hat{j} + \phi_{if}\,\hat{k}) \\
&= y_\phi\,\phi_{if}\,\hat{i} - x_\phi\,\phi_{if}\,\hat{j} + (x_\phi\,y_{if} - y_\phi\,x_{if})\hat{k}.
\end{aligned}$$

Let the normalized form of this vector be $\hat{V}_n = \bar{x}_n\,\hat{i} + \bar{y}_n\,\hat{j} + \bar{\phi}_n\hat{k}$. These vectors are shown in Fig. 2.18a. Now, a line (ℓ_b) parallel to the vector $\hat{V}_n$ is drawn at b and the points on this line, that have one of the allowable angles of rotation ϕ_λ are marked. If

for some such point, the corresponding angle is ϕ_{μ} then the distance of this point from b would be $t_{\mu} = (\phi_{\mu} - \phi_{b})/\overline{\phi}_{n}$. The x,y coordinates for this point (b_{μ}) would then be

$$x_{\mu} = x_{b} + t_{\mu} x_{n} \quad \text{and} \quad y_{\mu} = y_{b} + t_{\mu} y_{n}.$$

Now a plane (P_{μ}) parallel to the x-y plane is drawn at $\phi = \phi_{\mu}$. The intersections of this plane with each one of the obstacles would be nothing but the slices of those obstacles at $\phi = \phi_{\mu}$. Next, a line $(p_{\mu 1}\ q_{\mu 1})$ is drawn at b , on the plane P_{μ}. A search for points $C_{\mu 1}$ and $C_{\mu 2}$ is now made, along the line $p_{\mu 1}\ q_{\mu 1}$ exactly in the way explained in Fig. 2.11. The point b in Fig. 2.11 corresponds to b_{μ} here, the line pq to $p_{\mu 1}\ q_{\mu 1}$ and the obstacles O_1, O_2, O_3 and O_4 to the slices of these obstacles at $\phi = \phi_{\mu}$. The points C_1 and C_2 obtained from Fig.2.11, correspond to the $C_{\mu 1}$ and $C_{\mu 2}$ in this case.

This is repeated with M different lines $p_{\mu i}\ q_{\mu i}$, $i = 1,M$, symmetrically placed on the plane P_{μ}, about the point b_{μ}, as shown in Fig. 2.18b. Two points $C_{\mu(2i-1)}$ and $C_{\mu(2i)}$ are obtained for each one of these lines. All the $C_{\mu i}$'s obtained constitute the possible intermediate points on the plane P_{μ}. The intermediate points for all the planes P_{μ}, $\mu = 1,N$ where N is the number intervals into which the rotation range ϕ_{min}- ϕ_{min} has been divided. This way we get MN possible intermediate points, M on each plane P_{μ}.

Now, one of these MN points is to be chosen as the intermediate point. This choice is again based on the A^* algorithm,

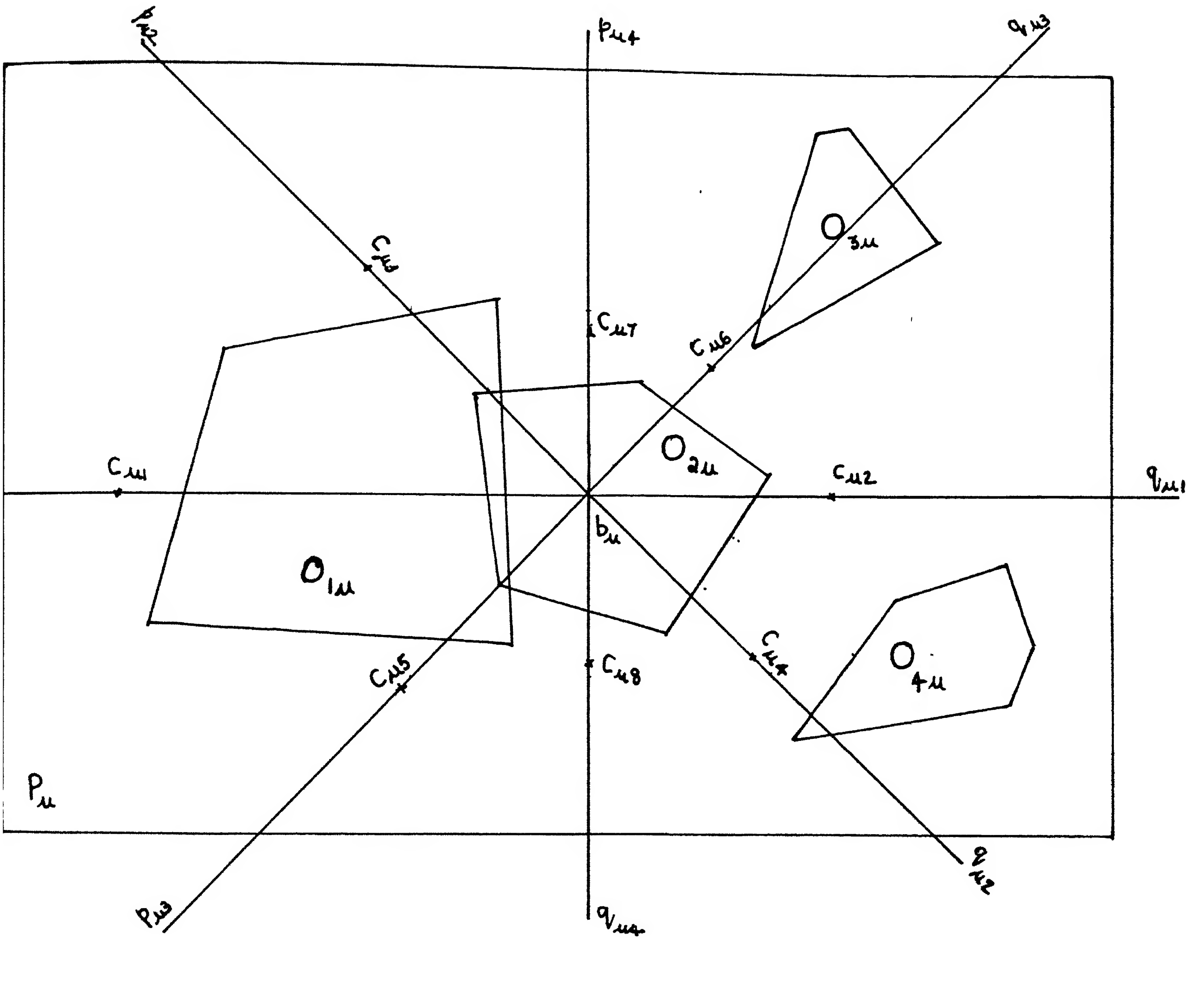


Fig. 2.18b Location of the intermediate points $C_{\mu j}$ on the ~~plane~~ P_{μ}.

as it was in the case of a circular object. The heuristic ordering function used here is again the distance of the intermediate points from the point b . The distance is calculated as follows. Let the distance of a point $C_{\mu i}$ from the corresponding b_μ be $t^{\phi}_{\mu i}$. The distance of b_μ from b is t_μ. The former distance is along $\hat{V}_n$ and the latter is along the plane P_μ. Since $\hat{V}_n$ is normal to P_μ, the distance $t_{\mu i}$ of $C_{\mu i}$ from b would be given by,

$$t_{\mu i} = [(t_\mu)^2 + (t^{\phi}_{\mu i})^2]^{1/2}$$

These $t_{\mu i}$'s serve the same role as did the t_{c1} and t_{c2} in Fig. 2.14. Here too, some of $C_{\mu i}$'s can be discarded based on conditions similar to those mentioned in Section 2.2. .. The only difference would be that in Fig. 2.14, the line pq would now be replaced by the plane P_n. The rest of the search is exactly the same as the recursive search described in Sec.2.2. .. Here too, the path is refined at every stage, to finally yield the optimum path.

The algorithm described above for find path with rotation will work as such only for objects which are convex. This poses no serious limitations however, since any non-convex polygon can be broken up into convex sub-polygons. Let the number of convex sub-polygons into which the object is broken up be N_c. Envelopes are generated around an obstacle with each one of these N_c convex polygons. These N_c envelopes are now treated

as N_c separate obstacles. So, if the number of obstacles in the original problem were ONUM, we end up with N_c x ONUM obstacles, all of which have been formed with convex polygonal objects. The rest of the algorithm is the same as it was for convex polygonal objects.

The implementations of these algorithms are discussed in the following section.

CHAPTER III

IMPLEMENTATION OF THE ALGORITHM

This section describes briefly, the implementation details of the algorithm based on the configuration space approach. The first problem that is to be handled during implementation is, the efficient representation of a polygon. Here the geometrical shape of the polygon itself suggests the appropriate data structure i.e. a circular linked list. Each node of the list is chosen to represent a vertex of the polygon.

Referring to Fig. 3.1, a reference point R is first chosen for every polygon. This choice can be arbitrary, except that the point must be within the polygon.

A local frame of reference $x_l - y_l$ is fixed to the object, with its origin at the reference point. The position of the object can now be completely specified by giving the (x_R, y_R) coordinates (with respect to the global frame x-y) of the reference point and the orientation ϕ of the frame $x_l - y_l$ with respect to the global x-axis. The polygon is now represented by a header, as in Fig. 3.2, with three fields, one each for x_R, y_R and ϕ. Here by reading the values of these three fields, the position of the polygon will be completely known.

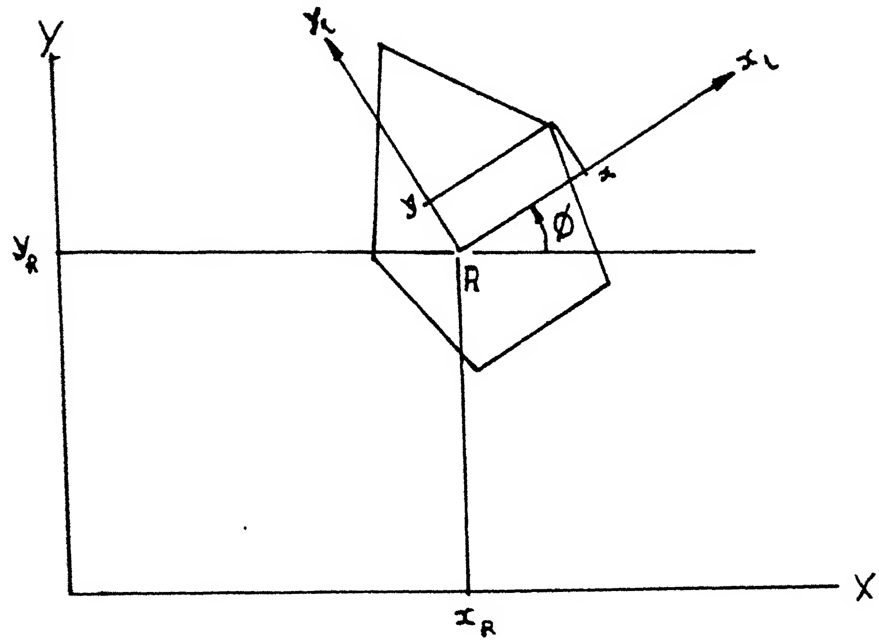


Fig. 3·1 Representation of a Polygon

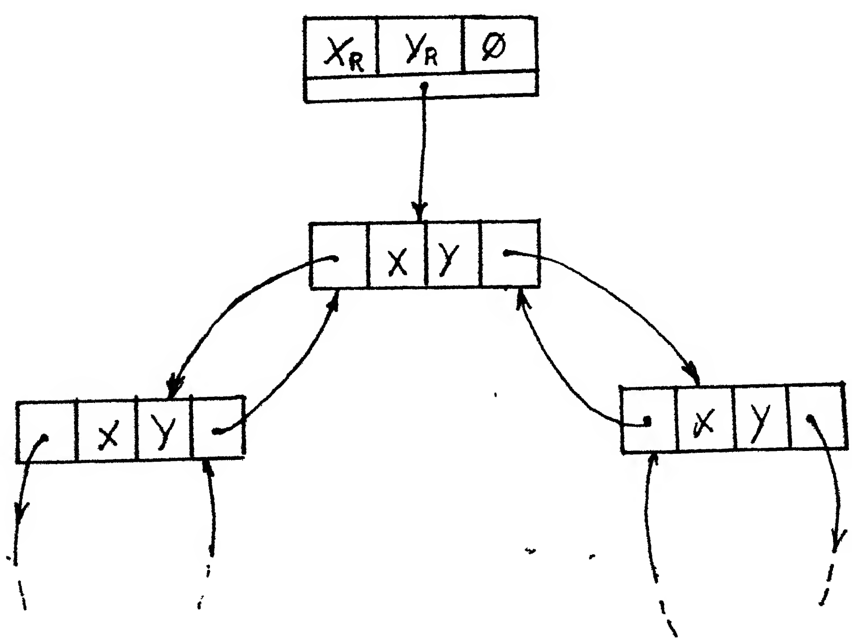


Fig 3·2 Data Structure for representing a Polygon.

Now the vertices of the polygon are described with respect to the local frame, as x, y coordinates. A node is now created for every vertex of the polygon, with each node having two fields, one each of x and y. All these are linked in both clockwise and the counter-clockwise directions around the reference point, in the same order as the vertices. The header inturn points to some node in the list.

With this representation, the coordinates in the global frame of any vertex of the polygon are determined as follows: Let the header read (x_R, y_R, ϕ). Therefore, as in Fig. 3.3, the coordinates of the reference point R of the polygon is at (x_R, y_R) with respect to the global frame G and the local frame is tilted at an angle ϕ to G. Now the x and y fields of the node corresponding to the vertex is read. Let these be x and y respectively. So we have,

$$r = \sqrt{x^2 + y^2} \quad \text{and} \quad \mathrm{Tan}\ \gamma = y/x$$

$$\text{or} \quad \gamma = \mathrm{Tan}^{-1}\ (y/x).$$

The global coordinates of the vertex can now be determined as

$$x_g = x_R + r \cos\ (\phi + \gamma)$$

and $\quad y_g = y_R + r \sin\ (\phi + \gamma)$ where (x_g, y_g) are the coordinates, in the global frame, of the vertex.

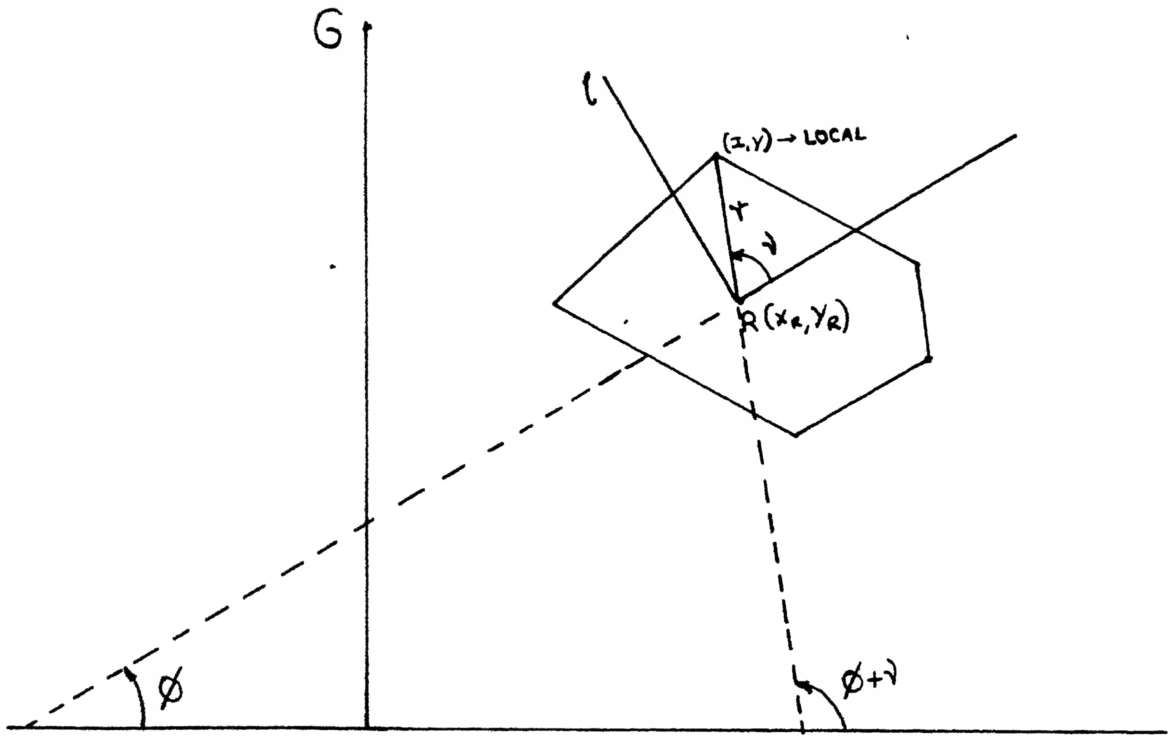


Fig. 3.3 Determination of the Global coordinates of the Vertices of a Polygon

The object and the obstacles are all represented as above. The headers of the all the obstacles together, are stored in an array.

In the case of findpath with rotation, the slice at every allowable ϕ_λ as defined in Sec. 2.3.2, of the 3D-obstacle, is a polygon. Since the maximum number of vertices that any of these slices can have is the sum of the number of vertices of the object and the 2D-obstacle, a circular list of length equal to the sum of these two is created. The x, y fields of these nodes are now made one-dimensional arrays of size N, where N is the number of slices. So that the $(x(\mu), y(\mu))$'s of all the nodes taken together would represent the slice at ϕ_μ.

The next problem to be dealt with is that of checking if a given line intersects an obstacle, and if it does then to determine the points of intersection. Considering only the 2-D case i.e., the case discussed in Sec. 2.2. , let the given line be from (x_1, y_1) to (x_2, y_2). The parametric equation of the line passing through these two is,

$$x = x_1 + t\,(x_2 - x_1)$$

$$\text{and,} \quad y = y_1 + t\,(y_2 - y_1).$$

Now an edge $\overline{V_iV_j}$ of the obstacle is chosen. Let the coordinates of the vertices V_i and V_j be (x_i, y_i) and (x_j, y_j) respectively.

The parametric equation for this edge is,

$$x = x_i + t_1\,(x_j - x_i)$$

$$\text{and,} \quad y = y_i + t_1\,(y_j - y_i)$$

So, for the point of intersection of the edge and the line, we have,

$$x_1 + t(x_2 - x_1) = x_i + t_1 (x_j - x_i)$$

and

$$y_1 + t(y_2 - y_1) = y_i + t_1 (y_j - y_i)$$

These two equations are now solved for t and t_1. The intersection of these two lines will matter only if the point of intersection lies within both the segments, or in other words, both t and t_1 must lie between 0 and 1.

$$\text{i.e.} \quad 0 \leq t \leq 1 \quad \text{and} \quad 0 \leq t_1 \leq 1.$$

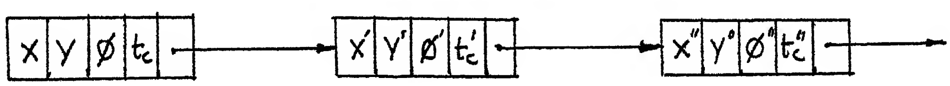
So, t and t_1 are checked forthese conditions. If these are satisfied, then the value of t is recorded, otherwise it is ignored. This is carried out for every edge of the obstacle. If none of the edges of the obstacle intersect the line joining (x_1, y_1) and (x_2, y_2) subject to the conditions specified, then obviously the line does not cut the obstacle. If there are edges which intersect this line, then the two extreme values (lowest and highest) of t obtained in the process are noted as t_a and t_b. It follows that the points of intersection corresponding to these two values of t are (x_a, y_a) and (x_b, y_b) where,

$$x_a = x_1 + t_a (x_2 - x_1), \quad y_a = y_1 + t_a (y_2 - y_1)$$
$$x_b = x_1 + t_b (x_2 - x_1), \quad y_b = y_1 + t_b (y_2 - y_1).$$

For the problem involving rotation of the object, checking for intersections of a line joining (x_i, y_i, ϕ_i) to (x_f, y_f, ϕ_f), with any of the grown 3-D obstacles, as explained in Section 2.3.2,

is based on checking if some point $(x_\omega, y_\omega, \phi_\omega)$ on this line is inside a polygon (slice of the obstacle at ϕ_ω) or not. This is done by first joining the reference point of the polygon to be given point by a straight line. This line is now checked for intersections with the edges of the polygon. If there are no such intersections, then the point is inside the polygon, otherwise, it is outside.

The A^* algorithm is implemented by maintaining an ordered list of valid intermediate points for every stage of the recursion. This could be an ordinary linked list.



Each node of the list represents ene intermediate point and contains values of the x-coordinate, y-coordinate, orientation and the value of the ordering function of the intermediate point. First of these nodes is chosen for further path search. If the path is found to be infeasible, then this node is deleted and the path search tried out with the next node in the list. This is repeated until a successful path is found. At this, the values of the x, y, and ϕ fields of the successful node are recorded and the whole list is disposed off. If the list becomes empty before a successful path has been found, then it means that a path from (x_i, y_i, ϕ_i) to (x_f, y_f, ϕ_f) does not exist.

The rest of the implementation involves the use of a procedure which would call itself recursively to determine the path. Every level of recursion, contains an ordered list of intermediate points as explained above, and the list is disposed off while popping out of this level. A path list is also maintained, so that whenever a straight line path to a new point is found from the point which is the last in the list, the new point gets appended to this list. The first element of this list is obviously the starting point and the search ends when the destination point gets appended to this list. Before popping out of every recursion level, the path is refined in the manner discussed in Sec. 2.2. . The path list generated, is used for doing this.

This implementation was done entirely in Pascal because of the complicated data structures that were necessary.

The actual program codes, implementing the penalty function algorithm and the configuration space algorithm are given in Appendix A and Appendix B respectively. **A procedural level skeleton of the algorithm implementation for the configuration space approach, is given in Appendix C.**

CHAPTER IV

RESULTS AND DISCUSSIONS

The implementation of the algorithms proposed was done on a ND-560 supermini computer and the graphic display was got on Tektronix-4109 terminals. The results obtained are presented below.

4.1 Penalty Function Approach

4.1.1 Circular Object

The solution obtained, for a sample problem is shown in Fig. 4.1 alongwith the obstacles. The plots of the joint angles θ_i, $i = 1,4$ with respect to the path length is shown in Fig. 4.2(a) through 4.2(d). The lengths of the links ℓ_1, ℓ_2, ℓ_3 and ℓ_4 as shown in Fig. 2.4 are 210.0, 105.0, 125.0 and 55.0 mm respectively. The limits on the joint angles θ_1, θ_2, θ_3 and θ_4 are:

(i) $\theta_{1\ max} = 94.6^\circ$, $\theta_{1\ min} = -114.6^\circ$

(ii) $\theta_{2\ max} = 114.6^\circ$, $\theta_{2\ min} = -10.0^\circ$

(iii) $\theta_{3\ max} = 106.5^\circ$, $\theta_{3\ min} = -106.5^\circ$

(iv) $\theta_{4\ max} = 119.7^\circ$, $\theta_{4\ min} = -114.6^\circ$

The cpu time taken in this case is 4.9 sec.

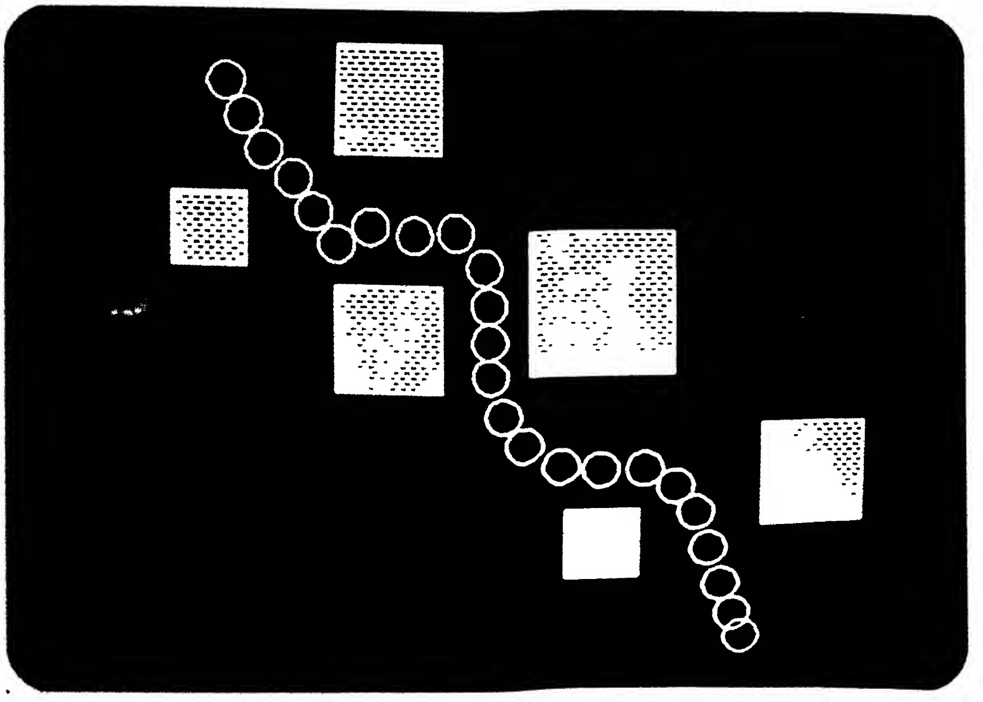


Fig. 4·1

FIG. 4.2 Plot of the Joint Angles vs Path Length

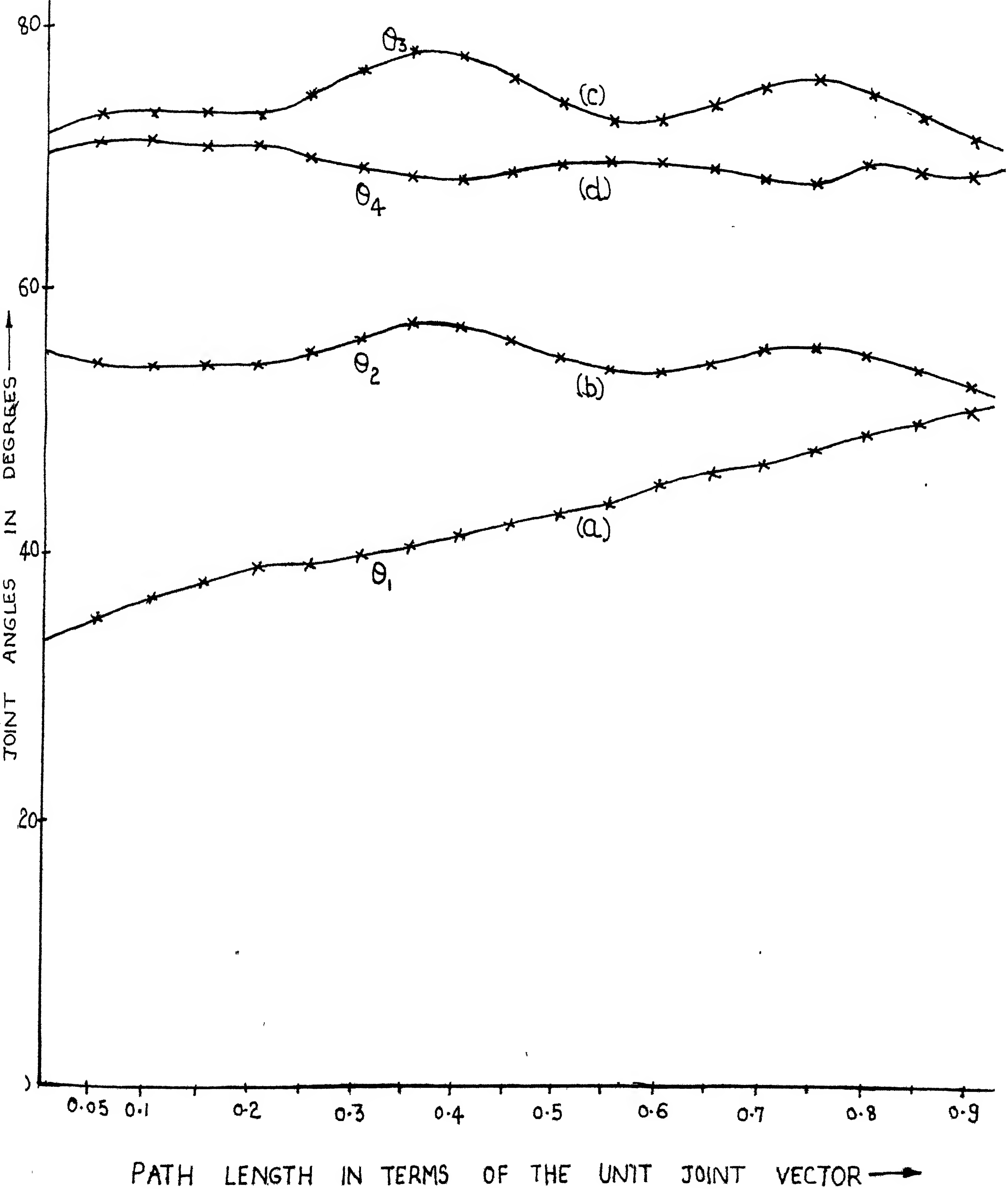


FIG. 4.2 Plot of the Joint Angles vs Path Length

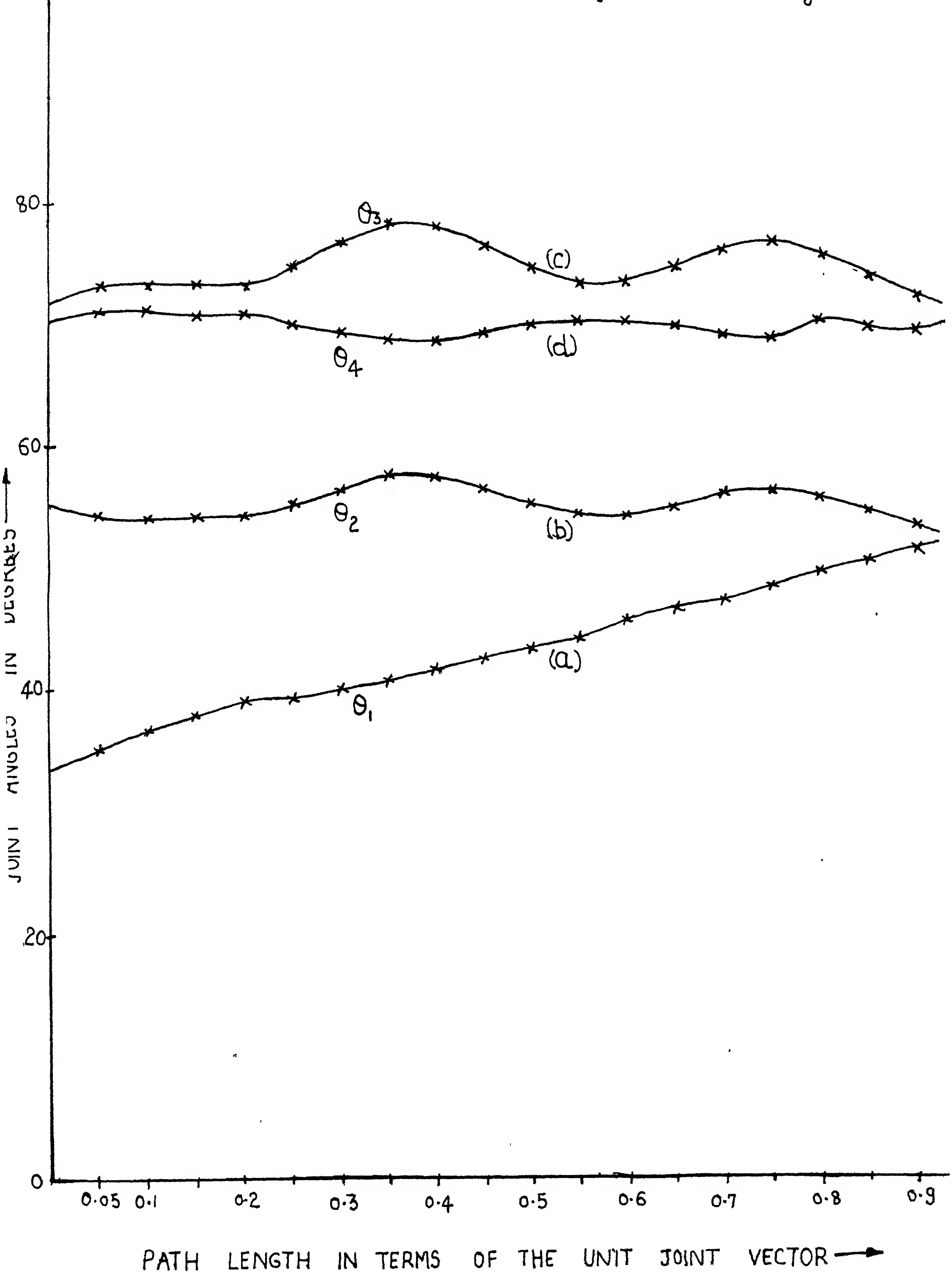


Fig. 4.3 shows a case where the algorithm failsto work. This is because of a local minimum that is present at the plane where the object can be seen to have got stuck in Fig. 4.3. Since a minimum has been reached, the object does not proceed any further.

4.1.2 Polygonal Object

A solution obtained in this case is shown in Fig. 4.4. The plots of the joint angles θ_i, $i = 1,5$ in this case are shown in Figures 4.5(a) through 4.5(e). The limits on θ_5 are: $\theta_{5\ max} = 119.7^\circ$ and $\theta_{5\ min} = -114.5^\circ$. The cpu time taken in this case is 48.6 sec.

Fig. 4.6 shows a case, where the algorithm fails. The reason is again the presence of local minima. It was observed by studying several cases, that the proposed algorithm works well even if the obstacles are closely cluttered, though the problem of local minima still remains. Hence, the above algorithm can be combined very profitably with global search technique, where some intermediate targets could be generated.

4.2 Configuration Space Approach:

The algorithm proposed under this approach, is applicable only for cartesian robots. The aim here is only to obtain a path for the object, that avoids collisions with any of the obstacles.

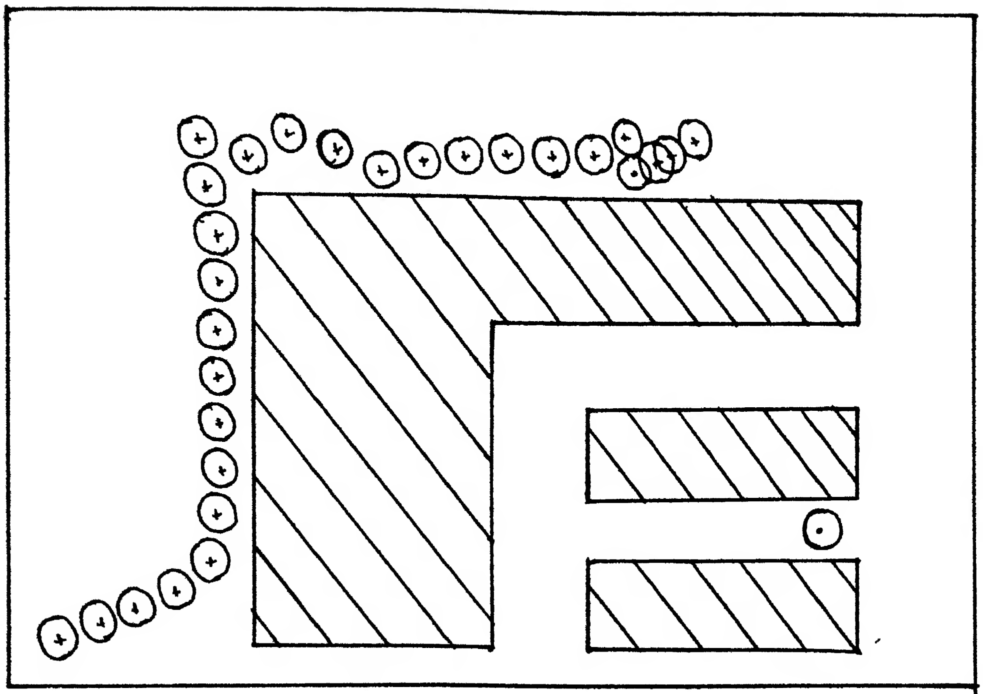


Fig. 4.3

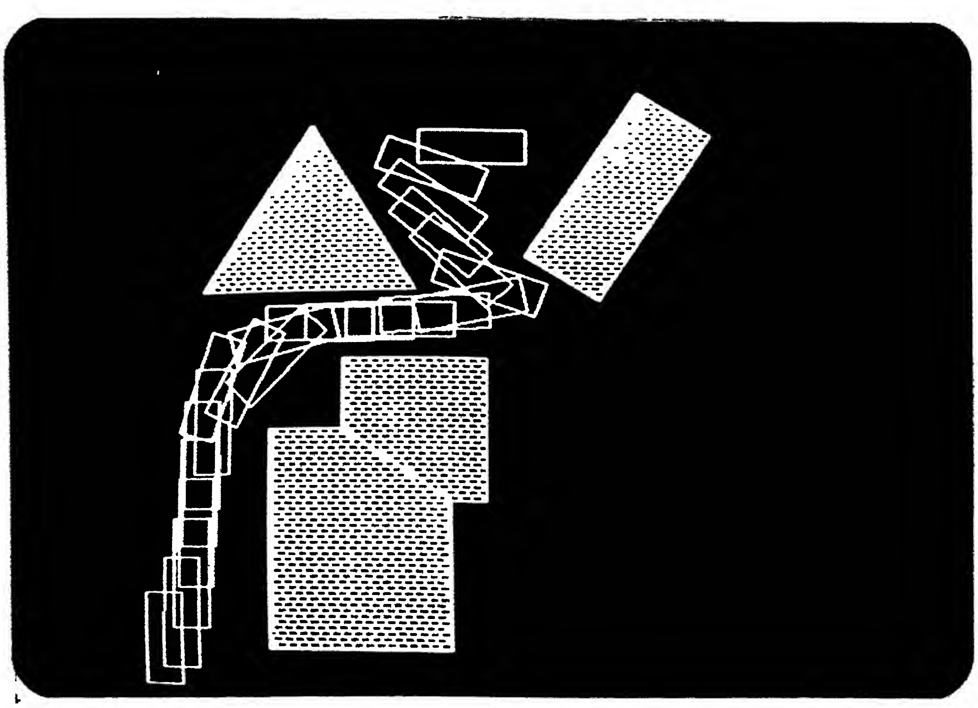
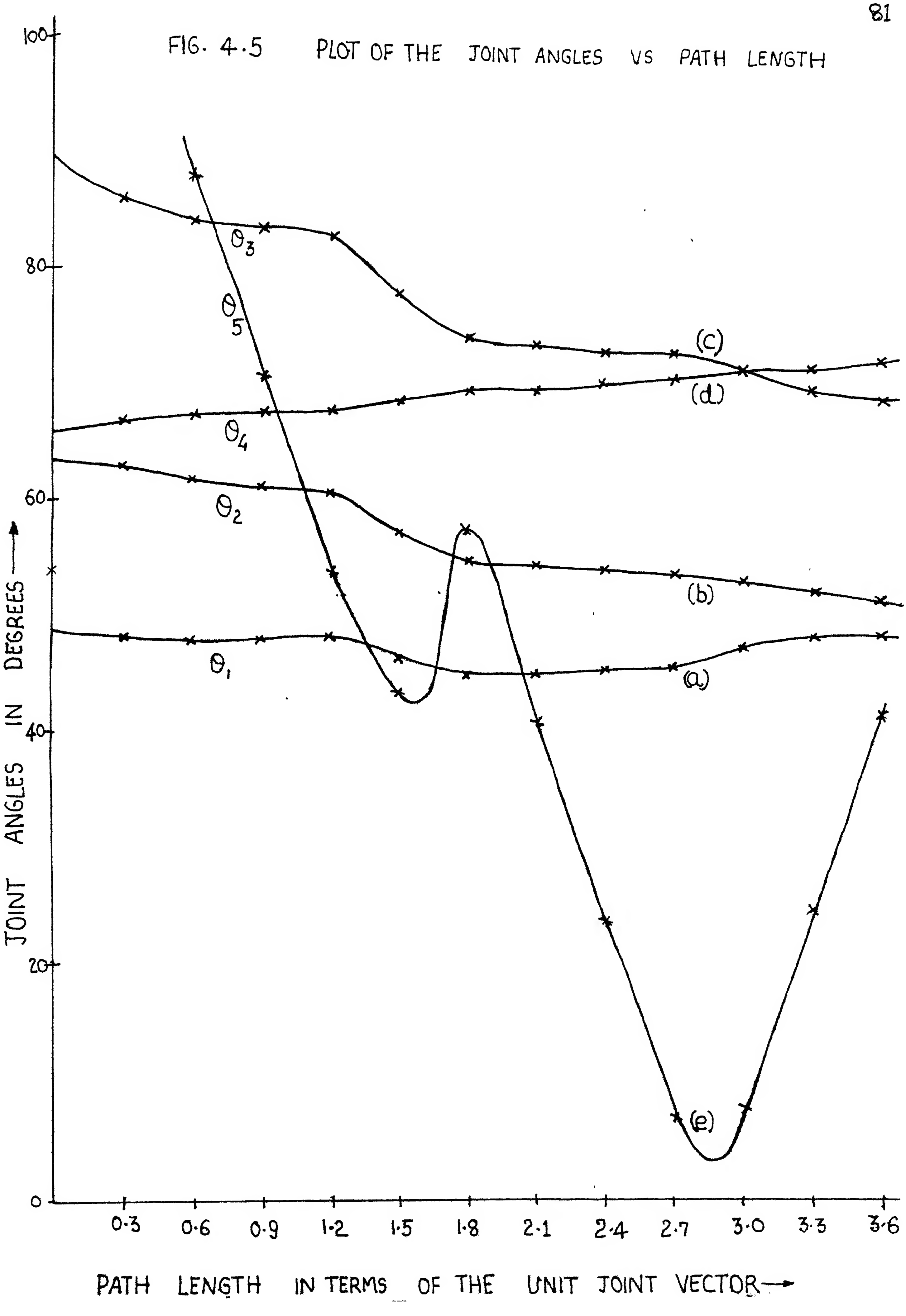


Fig. 4.4

FIG. 4.5 PLOT OF THE JOINT ANGLES VS PATH LENGTH



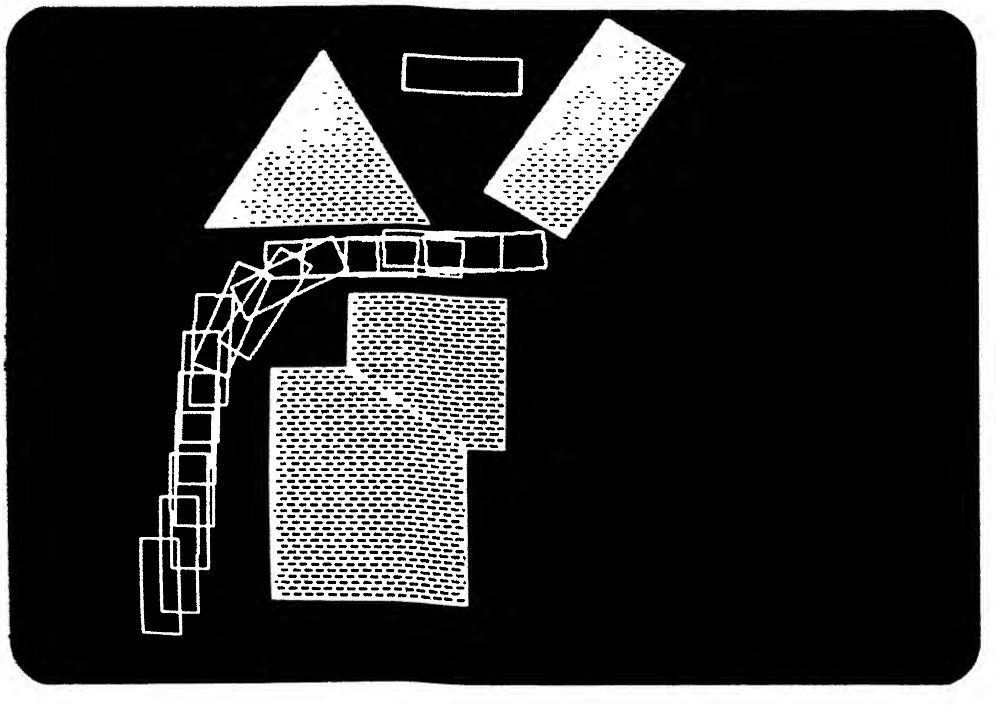


Fig. 4.6

4.2.1 Generation of Envelopes:

Figures 4.7 and 4.8 show the envelopes generated around a polyhedral obstacle for two different orientations of a rectangular object. The envelopes have been shown by dotted lines. The algorithm for generating these envelopes has been discussed in Sec. 2.3.

4.2.2 Circular Objects:

The path obtained for the case in Fig. 4.3, using this algorithm is shown in Fig. 4.9. The path shown is the final path which is obtained after refinements as discussed in Sec.2.2. The cpu time for this case is 1.1 sec.

4.2.3 Polygonal Objects:

Figures 4.10 and 4.11 show the final paths generated by this algorithm for two cases where the one in Fig. 4.10 corresponds to the one in Fig. 4.6. As can be seen from Fig. 4.6, the penalty function method failed in this case. This particular problem has also been solved by Lozano-Perez and R.A. Brooks [6] the solution for which they have shown in their paper. Their running time was of the order of 10 minutes of wall clock time on a single user MIT Lisp machine. According to them, this problem is one of the toughest findpath problems solved by any program. The running time of this problem in our case is 4 min 22.8 sec. on ND-560 time shared machine with 7 terminals.

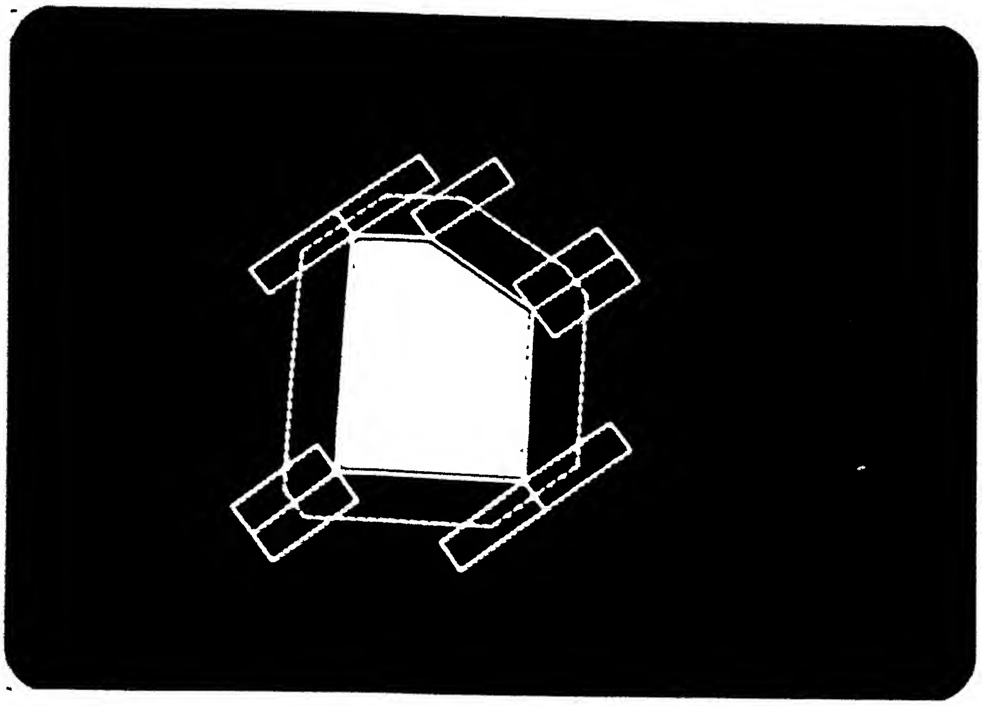


Fig. 4.7

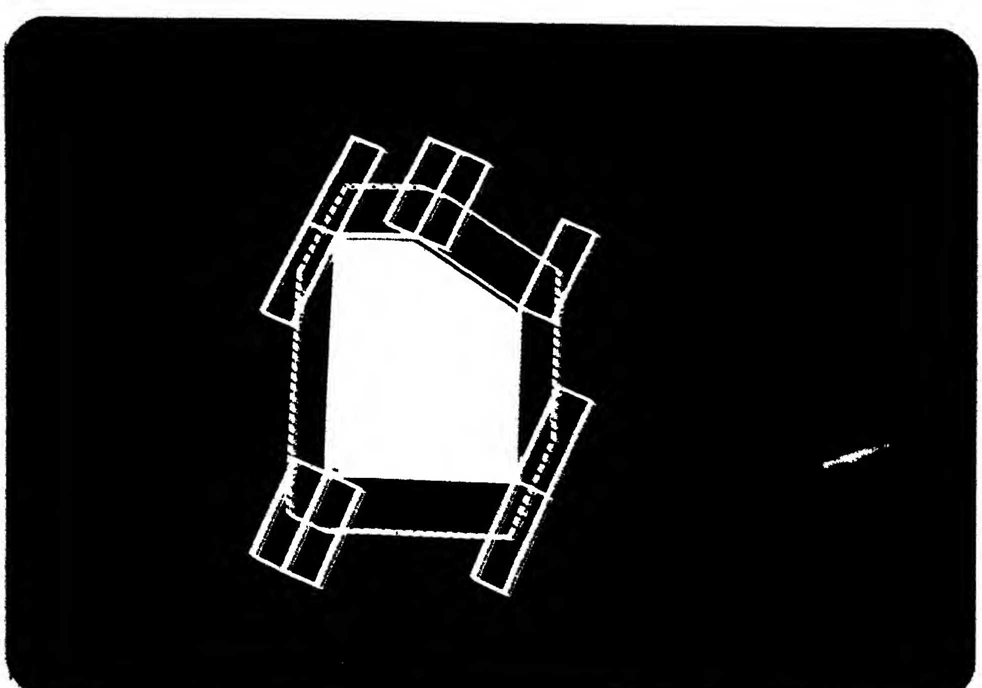


Fig. 4.8

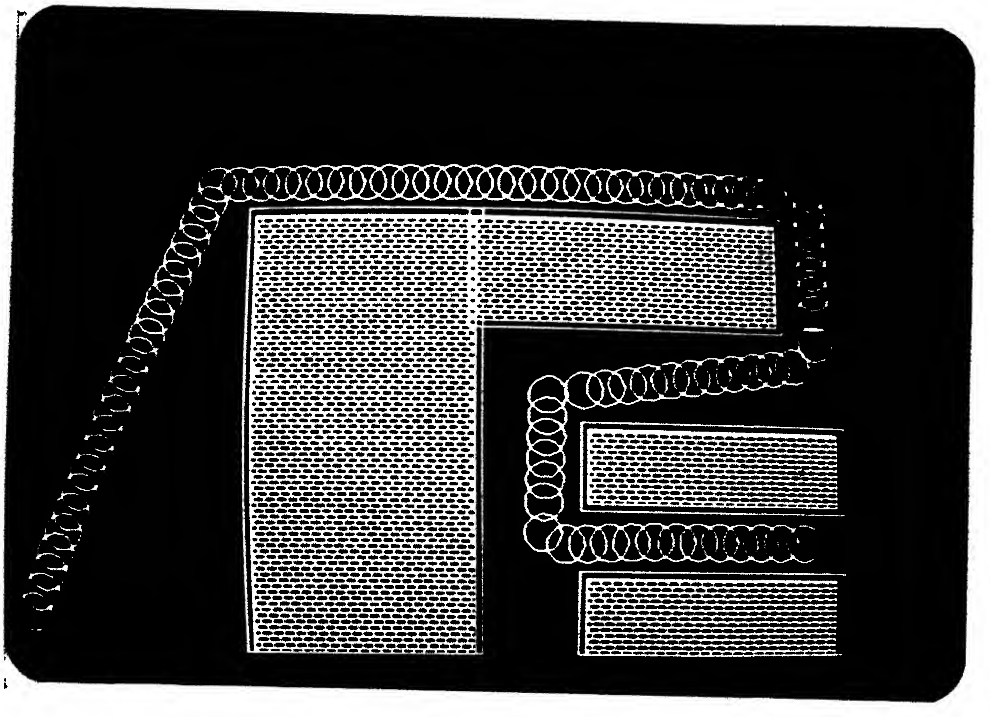


Fig. 4·9

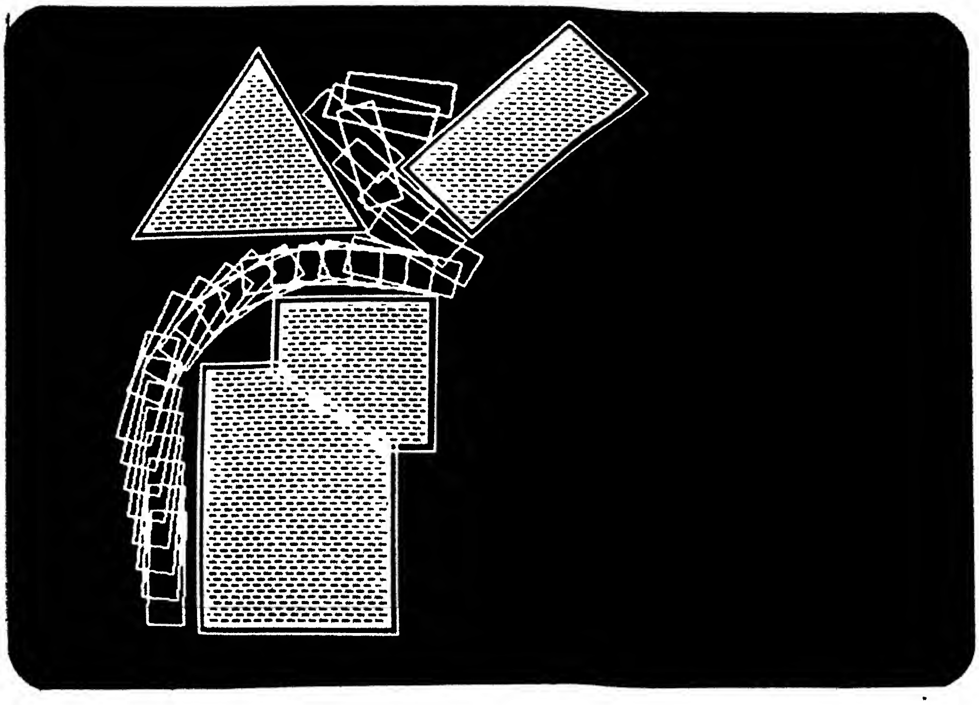


Fig. 4.10

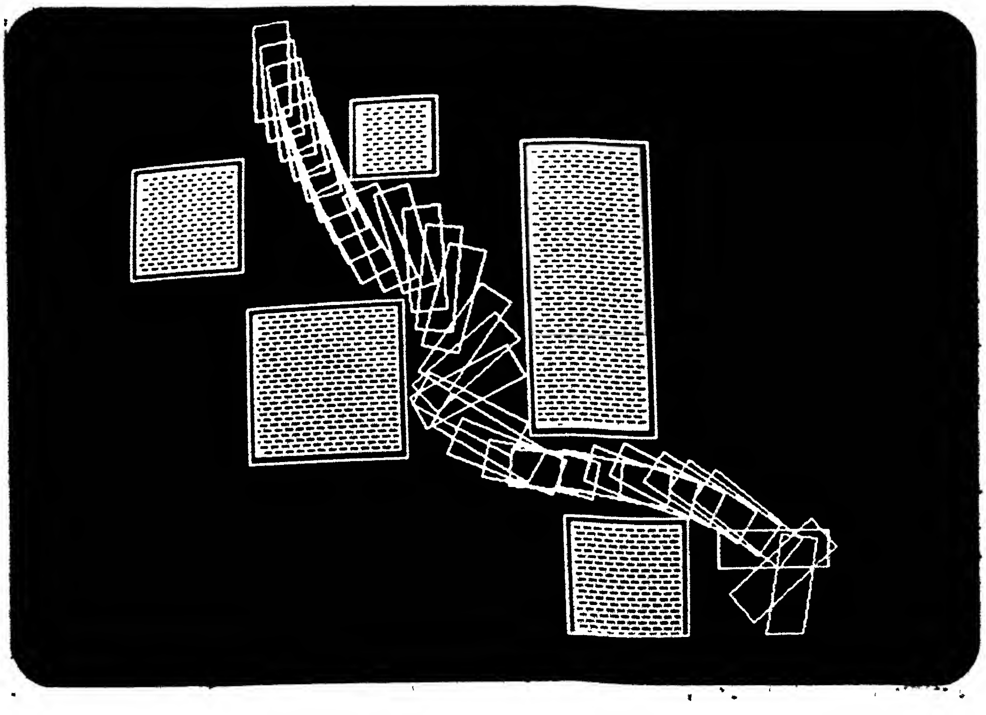


Fig. 4.11

CHAPTER V

CONCLUSIONS

The two new algorithms for find path problem have been proposed in this work. The first algorithm is based on Penalty Function Approach . The proposed method of working in joint coordinates directly is shown to work well and offers the advantage of taking care of the limitations on the joint rotations simultaneously while finding the collision free path. Moreover, since there are no approximations on the shapes of object or obstacles, the obtained path needs no further checks or modification. The algorithm is not suitable in dealing with the cases, where the object has to circumvent a relatively long obstacle. Otherwise, the proposed algorithm is successful even if the work space is thickly populated with obstacles.

The second algorithm is proposed along the lines of Free Space Approach . The suggested method of concentrating on the free space only around the obstacles is found to be successful in all cases tried, even with the long obstacles, where the first algorithm failed. With the second algorithm, the objective was only to find a collision free path. In this work, no attempt has been made to solve the inverse kinematic problem while using the second algorithm. This algorithm, though, is very useful for mobile autonomous robots which gather the workspace information through vision systems.

BIBLIOGRAPHY

1. Shimon Y. Nof, Handbook of Industrial Robotics, John Wiley and Sons, New York, 1985.

2. Boyse, J.W., Interference Detection among Solids and Surfaces, Comm. of the ACM, Vol. 22, No. 1, 1979, pp. 3-9.

3. Khatib, O., Commande Dynamique dans l'espace operationnel des robots manipulateurs en presence d'obstacles, Docteur Ingenieur Thesis, l'Ecole Nationale Superieure de l'Aeronautique et de l'Espace, Toulouse, France, 1980.

4. Lozano-Perez, T., Spatial Planning: A Configuration Space Approach, IEEE Transactions on Computers, C-32, No. 2, Feb. 1983.

5. Rodney A, Brooks, Solving the Find-Path Problem by Good Representation of Free Space, IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-13, No. 3, March/April, 1983.

6. Rodnay A. Brooks and Tomas Lozano-Perez, A Subdivision Algorithm in Configuration Space for Findpath with Rotation, IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-15, No. 2, March/April, 1985.

7. Richard P. Paul, Robot Manipulators: Mathematics - Programmi and Control, The MIT Press.

8. Elmer G. Gilbert and Daniel W. Johnson, Distance Functions and their Application to Robot Path Planning in the Presence of Obstacles, IEEE Journal of Robotics and Automation, Vol. RA-1, No. 1, March, 1985.

9. N.J. Nilsson, Problem-Solving Methods in Artificial Intelligence, New York, McGraw Hill, 1971.

APPENDIX A

PROGRAM IMPLEMENTING THE PENALTY FUNCTION ALGORITHM

```
program avoidobstacle(input,output);
 label
      100;
  const
*******************************************************************
 convert--conversion factor from radians to degrees
 n--dimensionality of the reduced joint vector
 minct,maxct--arbitrary constants
*******************************************************************
      convert=57.29577951;
      n=3;gra=68.2667;nstep=5;
      minct=1.0e-4;maxct=1.0e+5;
      pi=3.1415927;
  type
      vector=array [1..n]of real;
      obstset=set of 1..10;
*******************************************************************
 pointer--node representing a vertex of a polygon}
*******************************************************************
      pointer=^node;
      node=record
                 x,y,theta,l:real;
                 next,prev:pointer
           end;
**********************************************************************
 pointerp--nodes representing the centers of the hypothetical inscribing
          circles
**********************************************************************
      pointerp=^nodep;
      nodep=record
                 r,theta,rad:real;
                 next:pointerp
           end;
**********************************************************************
 pspolygon--header node representing a pseudo-polygon formed by the centers
           of the hypothetical inscribing circles
**********************************************************************
      pspolygon=^lframep;
      lframep=record
                 x,y,theta:real;
                 next:pointerp
            end;
**********************************************************************
 polygon--header node representing a polygon
**********************************************************************
      polygon=^lframe;
      lframe=record
                 x,y,theta:real;
                 next:pointer
            end;
  var
     grad,jc,jcf,jc1,s:vector;
     llim,ulim,fact:array [0..4] of real;
     phi,xf,yf,fphi,xi,yi,iphi,l0,l1,l2,l3,step,gstep,y0,slope,factor,c0,
     c1,c2,a,b,c,x1,y1,rotfact1,rotfact2,rotfact3,slperr,disterr,dist1,dist2,
     s1,xg,yg,rotg,approach:real;
     i,j,nvert,onum,ox,oy,p,count,npseuds:integer;
     obstacle:array [1..10] of polygon;
     vertex1:pointer;object:polygon;psobject:pspolygon;
```

```
     clstobstacle:obstset;
     check:boolean;
     afile:text;
$include (pas)graphics:pas
******************************************************************************
 'getinfo' computes the global coordinates'(x,y)' of the vertex 'vertex'
  which is in the polygon 'object'
******************************************************************************
  procedure getinfo(object:polygon;vertex:pointer;var x,y:real);
        var
             x1,y1,x2,y2,r,theta:real;
        begin
             x1:=object^.x;y1:=object^.y;
             theta:=vertex^.theta+object^.theta;
             x2:=vertex^.x;y2:=vertex^.y;
             r:=sqrt(x2*x2+y2*y2);
             x:=x1+r*cos(theta);
             y:=y1+r*sin(theta)
        end;
******************************************************************************
 'Psgetinfo' computes the global coordinates '(x,y)' of the centre of the
 hypothetical circle given by 'vertex' of the polygon 'object'.It also
 gives the radius 'rad' of the circle
******************************************************************************
  procedure psgetinfo(object:pspolygon;vertex:pointerp;var x,y,rad:real)
        var
             x1,y1,x2,y2,r,theta:real;
        begin
             x1:=object^.x;y1:=object^.y;
             theta:=vertex^.theta+object^.theta;
             r:=vertex^.r;
             rad:=vertex^.rad;
             x:=x1+r*cos(theta);
             y:=y1+r*sin(theta)
        end;
******************************************************************************
 'Linetopt' is the distance of the point '(xc,yc)' from the line joining
 the points '(xa,ya)' and '(xb,yb)'.Linetopt is assigned an arbitrarily larg
 value if perpendicular from the pt. to the line does not fall within the
 segment
******************************************************************************
  function linetopt(xa,ya,xb,yb,xc,yc:real):real;
        var
             l,d1,d2,cos1,cos2,a,b:real;
        begin
             l:=(xa-xb)*(xa-xb)+(ya-yb)*(ya-yb);
             d1:=(xa-xc)*(xa-xc)+(ya-yc)*(ya-yc);
             d2:=(xb-xc)*(xb-xc)+(yb-yc)*(yb-yc);
             if (abs(d1)<minct) or (abs(d2)<minct) then linetopt:=0.0
             else
                begin
                     cos1:=d1+l-d2;cos2:=d2+l-d1;
                     a:=2.0*sqrt(d1*l);b:=2.0*sqrt(d2*l);
                     if (abs(1.0-cos1/a)<minct) or (abs(1.0+cos1/a)<minct) th
                       linetopt:=0.0
                       else
                         begin
                              if (abs(cos1/a)<minct) then linetopt:=sqrt(d1)
                              else if (abs(cos2/b)<minct) then linetopt:=sqrt(
                              else
```

```
            move(x1,y1);
            repeat
                  vertex:=vertex^.next;
                  getinfo(obj,vertex,x,y);
                  x1:=trunc((x-100.0)*gra);y1:=trunc((y-100.0)*gra);
                  if (x1<0) then x1:=0;if (y1<0) then y1:=0;
                  if (x1>4090) then x1:=4090;if (y1>3080) then y1:=3080;
                  draw(x1,y1)
            until (vertex=obj^.next)
      end;
{*********************************************************************
 Jointspace--inverse transformation from cartesian coord.'(x,y,phi)' to
 the coord. given by the vector 'jc'
*********************************************************************}
  procedure jointspace(x,y,phi:real;var jc:vector);
      var
          d,d1,d2,a,b,c,a1,a2,a3,theta0,theta1,theta2,theta3:real;
      begin
          d:=x*x+y*y;
          d1:=l3-l0+y0;d2:=d1*d1;
          theta2:=arccos((d+d2-l2*l2-l1*l1)/(2.0*l1*l2));
          a:=l2*cos(theta2)+l1;b:=l2*sin(theta2);c:=sqrt(d);
          a1:=a*a+b*b;a2:=c*a;a3:=c*c-b*b;
          a:=sqrt(a2*a2-a3*a1)/a1;b:=a2/a1;
          a1:=b-a;a2:=b+a;
          if (a1<0.0) then theta1:=arccos(a2) else theta1:=arccos(a1);
          theta3:=theta1-theta2+1.57079633;
          theta0:=arctan(y/x);
          jc[1]:=theta0;jc[2]:=theta1;jc[3]:=phi-jc[1]+1.57079633;
      end;
{*********************************************************************
 Given the joint vector 'point','cartes' computes the cartesian coord.
 '(x,y,phi)' and the joint angles 'theta2' and 'theta3'
*********************************************************************}
  procedure cartes(point:vector;var x,y,phi,theta2,theta3:real);
      var
          theta0,theta1,a1:real;
      begin
          theta0:=point[1];theta1:=point[2];phi:=point[3];
          theta3:=arccos((l1*sin(theta1)-l3+l0-y0)/l2);
          theta2:=theta1-theta3+1.57079633;
          a1:=l2*sin(theta3)+l1*cos(theta1);
          x:=a1*cos(theta0);y:=a1*sin(theta0)
      end;
{*********************************************************************
 'Funval' is the value of the objective function at the point given by
 the joint vector 'point'
*********************************************************************}
  function funval(point:vector;sum:real):real;
      var
         theta:array [0..4] of real;
         a,d,x1,y1,sum1,sum2:real;
         i:integer;
     function distancefn(i:integer):real;
          var
             sum,x,y,r:real;
             vertex:pointerp;
{*********************************************************************
 'pdistancefn' is the dmin between the polygons 'object' and 'obstacle'
*********************************************************************}
```

```
                              if ((cos1>0.0) and (cos2>0.0)) then
                                linetopt:=sqrt(c2-cos2*cos2/(4*1))
                              else linetopt:=maxct
                         end
                   end
              end;
***************************************************************************
 'Max' checks if 't' is large (maxct) or not
***************************************************************************
function max(t:real):boolean;
   begin
       if (abs(t-maxct)<100.0) then max:=true
       else max:=false
   end;
******************************************************************************
 'Arccos' computes the angle whose cosine is 'cx'
******************************************************************************
function arccos(cx:real):real;
   var
       sx,a:real;
   begin
      if (abs(cx-1.0)<minct) then arccos:=0.0
        else
          if (abs(cx+1.0)<minct) then arccos:=pi
          else
            begin
                sx:=sqrt(1-cx*cx);
                if (abs(cx)>minct) then a:=arctan(sx/cx)
                else a:=pi/2.0;
                if (a<0.0) then a:=a+pi;
                arccos:=a
            end
   end;
procedure drawcircle(x,y,r:real);
   var
       x1,y1:integer;
       theta,dtheta:real;
   begin
       theta:=0.0;dtheta:=0.2;
       y1:=trunc((y-100.0)*gra);
       x1:=trunc((x+r-100.0)*gra);
       move(x1,y1);
       repeat
          theta:=theta+dtheta;
          x1:=trunc((x+r*cos(theta)-100.0)*gra);
          y1:=trunc((y+r*sin(theta)-100.0)*gra);
          draw(x1,y1)
       until (theta>=6.2831854)
     end;
  procedure drawpolygon(obj:polygon);
       var
           theta,x,y:real;
           x1,y1:integer;
           vertex:pointer;
      begin
          vertex:=obj^.next;
          getinfo(obj,vertex,x,y);
          x1:=trunc((x-100.0)*gra);y1:=trunc((y-100.0)*gra);
          if (x1<0) then x1:=0;if (y1<0) then y1:=0;
          if (x1>4090) then x1:=4090;if (y1>3080) then y1:=3080;
```

```
function pdistancefn(object,obstacle:polygon):real;
    label
        100;
    var
        xo,yo,phi1,phi2,xi,yi,xj,yj,l1,l2,oxi,oyi,oxj,oyj,d1,d2,d3,
        d4,d,dist,os:real;
        k,j:integer;
        vertex1,vertex2,overtex1,overtex2:pointer;
        sobject:array [1..2] of polygon;
        svertex:array [1..2] of pointer;
        xi1,xj1,yi1,yj1,l:array [1..2] of real;
        cw:array [1..2] of boolean;
        sea:boolean;
    procedure locsector(obt:polygon;phi:real;var vi,vj:pointer
                        var xi,yi,xj,yj:real);
        var
            iphi,jphi:real;
            vertex:pointer;
        begin
            vertex:=obt^.next;
            iphi:=vertex^.theta;
            jphi:=vertex^.prev^.theta;
            if (iphi<phi) and (jphi>phi) then
            repeat
                vertex:=vertex^.next;
                jphi:=vertex^.theta
            until (jphi>phi);
            vi:=vertex^.prev;
            vj:=vertex;
            getinfo(obt,vj,xj,yj);
            getinfo(obt,vi,xi,yi)
        end;
    function perpdist(d1,d2,l:real):real;
        var
            cos1,cos2,d:real;
        begin
            cos1:=d1+l-d2;cos2:=d2+l-d1;
            if (cos1>0.0) and (cos2>0.0) then
              begin
                d:=d2+l-d1;
                perpdist:=sqrt(d2-d*d/(4*l))
              end
             else perpdist:=-1.0
          end;
    procedure clstvert(i:integer);
        var
            j:integer;
            xi,yi,d,d1,d2:real;
            vertex:pointer;
        begin
            if (i=1) then j:=2 else j:=1;
            vertex:=svertex[i];
            if (cw[i]) then
              begin
                xi:=xj1[i];yi:=yj1[i]
              end
            else
              begin
                xi:=xi1[i];yi:=yi1[i]
              end;
```

```
            d:=dist;
            if (cw[i]) then d2:=ds else d1:=ds;
            repeat
             svertex[i]:=vertex;dist:=d;
             if (cw[i]) then ds:=d2 else ds:=d1;
             if (cw[i]) then
               begin
                    xj1[i]:=xi;yj1[i]:=yi
               end
             else
               begin
                    xi1[i]:=xi;yi1[i]:=yi
               end;
             if (cw[i]) then vertex:=vertex^.prev
             else vertex:=vertex^.next;
             getinfo(sobject[i],vertex,xi,yi);
             d1:=(xi-xi1[j])*(xi-xi1[j])+(yi-yi1[j])*(yi-yi1[j])
             d2:=(xi-xj1[j])*(xi-xj1[j])+(yi-yj1[j])*(yi-yj1[j])
             d:=perpdist(d1,d2,l[j])
            until (d=-1.0) or (d>dist)
        end;
begin
     xo:=obstacle^.x;yo:=obstacle^.y;
     if (abs(xo-x1))1.0e-3) then
       begin
            phi1:=arctan((yo-y1)/(xo-x1));
            if (xo<x1) then phi1:=phi1+3.1415927
            else
                 if (yo<y1) then phi1:=phi1+6.2831654
       end
     else
          if (yo<y1) then phi1:=4.71238899
          else phi1:=1.57079633;
     phi2:=phi1+3.1415927;
     if (phi2>6.2831654) then phi2:=phi2-6.2831654;
     phi1:=phi1-object^.theta;
     if (phi1<0.0) then phi1:=phi1+6.2831654;
     if (phi1>6.2831654) then phi1:=phi1-6.2831654;
     locsector(object,phi1,vertex1,vertex2,xi,yi,xj,yj);
     locsector(obstacle,phi2,overtex1,overtex2,oxi,oyi,oxj,oyj);
     l1:=vertex1^.l;l2:=overtex1^.l;
     d1:=(xi-oxi)*(xi-oxi)+(yi-oyi)*(yi-oyi);
     dist:=d1;
     d2:=(xi-oxj)*(xi-oxj)+(yi-oyj)*(yi-oyj);
     if (d2<dist) then dist:=d2;
     d3:=(xj-oxi)*(xj-oxi)+(yj-oyi)*(yj-oyi);
     if (d3<dist) then dist:=d3;
     d4:=(xj-oxj)*(xj-oxj)+(yj-oyj)*(yj-oyj);
     if (d4<dist) then dist:=d4;
     dist:=sqrt(dist);sea:=false;
     d:=perpdist(d1,d2,l2);
     if (d<>-1.0) and (d<dist) then
       begin
            sea:=true;
            cw[1]:=true;cw[2]:=false;
            ds:=d2;dist:=d;l[2]:=l2;
            svertex[1]:=vertex1;svertex[2]:=overtex2;
            xj1[1]:=xi;yj1[1]:=yi;
            xi1[2]:=oxi;yi1[2]:=oyi;
            xj1[2]:=oxj;yj1[2]:=oyj;
```

```
              sobjecr[1]:=object;sobject[2]:=obstacle
           end;
         d:=perpoint(d3,d4,l2);
         if (d<>-1.0) and (c<dist) then
           begin
                sea:=true;
                cw[1]:=false;cw[2]:=true;
                ds:=d3;dist:=d;l[2]:=l2;
                svertex[1]:=vertex2;svertex[2]:=overtex1;
                sobject[1]:=object;sobject[2]:=obstacle;
                xi1[1]:=xj;yi1[1]:=yj;
                xi1[2]:=oxi;yi1[2]:=oyi;
                xj1[2]:=oxj;yj1[2]:=oyj;
           end;
         d:=perpdist(d1,d3,l1);
         if (d<>-1.0) and (d<dist) then
           begin
                sea:=true;
                cw[1]:=true;cw[2]:=false;
                ds:=d3;dist:=d;l[2]:=l1;
                svertex[1]:=overtex1;svertex[2]:=vertex2;
                sobject[1]:=obstacle;sobject[2]:=object;
                xj1[1]:=oxi;yj1[1]:=oyi;
                xi1[2]:=xi;yi1[2]:=yi;
                xj1[2]:=xj;yj1[2]:=yj
           end;
         d:=perpdist(d2,d4,l1);
         if (d<>-1.0) and (d<dist) then
           begin
                sea:=true;
                cw[1]:=false;cw[2]:=true;
                ds:=d2;dist:=d;l[2]:=l1;
                svertex[1]:=overtex2;svertex[2]:=vertex1;
                sobject[1]:=obstacle;sobject[2]:=object;
                xi1[1]:=oxj;yi1[1]:=oyj;
                xi1[2]:=xi;yi1[2]:=yi;
                xj1[2]:=xj;yj1[2]:=yj
           end;
         if (sea) then
           repeat
              for k:=1 to 2 do
                begin
                  clstvert(k);
                  if (k=1) then j:=2 else j:=1;
                  if (cw[k]) then
                    begin
                         getinfo(sobject[k],svertex[k]^.prev,xi1[
                                  yi1[k]);
                         l[k]:=svertex[k]^.prev^.l;
                         d1:=sqr(xi1[k]-xj1[j])+sqr(yi1[k]-yj1[j]
                         svertex[k]:=svertex[k]^.prev;
                         xi1[j]:=xj1[j];yi1[j]:=yj1[j]
                    end
                  else
                    begin
                         getinfo(sobject[k],svertex[k]^.next,xj1[
                                 yj1[k]);
                         l[k]:=svertex[k]^.l;
                         d1:=sqr(xj1[k]-xi1[j])+sqr(yj1[k]-yi1[j]
                         svertex[k]:=svertex[k]^.next;
```

```
                        xj1[j]:=x11[j];yj1[j]:=y11[j]
                     end;
                    d:=perpdist(d1,ds,l[k]);
                    if (d<>-1.0) and (d<dist) then dist:=d else goto
                    100
                 end
               until (d=-1.0) or (d>dist);
               100:
              pdistancefn:=1/dist
          end;
****************************************************************************
 'cdistancefn' is the dmin between the circle of radius 'rc' with centre
 located at '(xc,yc)' and the polygon 'object'
****************************************************************************
     function cdistancefn(xc,yc,rc:real;object:polygon):real;
          var
             x1,y1,xj,yj,vdist,edist,a,b,c,d,cos1,cos2,ea:real;
             vertex,vertex1:pointer;
          begin
              vertex1:=object^.next;
              getinfo(object,vertex1,x1,y1);
              a:=(x1-xc)*(x1-xc)+(y1-yc)*(y1-yc);
              vdist:=a;edist:=1.0e+5;vertex:=vertex1;
              repeat
                   vertex:=vertex^.next;
                   getinfo(object,vertex,xj,yj);
                   b:=(xj-xc)*(xj-xc)+(yj-yc)*(yj-yc);
                   if (b<vdist) then vdist:=b;
                   c:=(xj-x1)*(xj-x1)+(yj-y1)*(yj-y1);
                   cos1:=a+c-b;cos2:=b+c-a;
              if (cos1>0.0) and (cos2>0.0) then
                begin
                    ea:=sqrt(b-cos2*cos2/(4*c));
                    if (ea<edist) then edist:=ea
                end;
              x1:=xj;y1:=yj;a:=b
        until (vertex=vertex1);
        a:=edist-rc;b:=sqrt(vdist)-rc;
        if (a<b) then c:=1/a else c:=1/b;
        if (c>0.0) then cdistancefn:=c else cdistancefn:=1.0e+5
    end;
         begin
             sum:=c0+pdistancefn(object,obstacle[1]);
             vertex:=psobject^.next;
             repeat
                   psgetinfo(psobject,vertex,x,y,r);
                   sum:=sum+cdistancefn(x,y,r,obstacle[1]);
                   vertex:=vertex^.next
             until (vertex=psobject^.next);
             distancefn:=sum
          end;
          begin
              d:=0.0;
              for i:=1 to (n-1) do d:=d+(point[i]-jcf[i])*(point[i]-jcf[i]
              if (sum<dist1) then d:=d+rotfact1*(point[3]-jcf[3])*
                                                (point[3]-jcf[3]);
              cartes(point,x1,y1,theta[4],theta[2],theta[3]);
              object^.x:=x1;object^.y:=y1;
              object^.theta:=point[3]+point[1]-1.57079633;
              psobject^.x:=x1;psobject^.y:=y1;
```

```
              psobject .theta:=object .theta;
              sum1:=0.0;
              if (check) then
                 for j:=1 to onum do
                    begin
                       a:=distancefn(j);
                       if (a>approach) then
                          begin
                             clstobstacle:=clstobstacle+[j];
                             sum1:=sum1+a
                          end
                       else clstobstacle:=clstobstacle-[j]
                    end
              else
                 for j:=1 to onum do
                    if (j in clstobstacle) then
                       sum1:=sum1+distancefn(j);
              sum1:=sum1+(1/(x1-100))+(1/(y1-100))+1/(150-y1)+1/(160-x1);
              theta[0]:=point[1];theta[1]:=point[2];
              sum2:=0.0;
              for j:=0 to 4 do
                begin
                   a:=(theta[j]-llim[j])/fact[j];
                   sum2:=sum2+(1/a)+(1/(1-a))
                end;
              funval:=d*(1+c1*sum1+c2*sum2)
           end;
***********************************************************************
 'norm' is the norm(length) of the vector 'x'
***********************************************************************
  function norm(x:vector):real;
       var
          i:integer;
          sum:real;
      begin
          sum:=0.0;
          for i:=1 to n do sum:=sum+x[i]*x[i];
          norm:=sqrt(sum)
      end;
*************************************************************************************
 'gradient'  computes the gradient 'grad' at the point given by the joint
  vector 'point'
*************************************************************************************
  procedure gradient(point:vector;var grad:vector;var slope:real);
       var
           i:integer;
           f1, f2, sum, fact:real;
       begin
           sum:=0.0;
           for i:=1 to (n-1) do sum:=sum+(point[i]-jcf[i])*(point[i]-jcf[i]);
           if (sum>dist2) then fact:=rotfact2 else fact:=rotfact3;
           for i:=1 to n do
                 begin
                      point[i]:=point[i]+gstep;
                      f1:=funval(point,sum);
                      point[i]:=point[i]-2*gstep;
                      f2:=funval(point,sum);
                      if (i<>3) then grad[i]:=(f1-f2)/(2*gstep)
                      else grad[i]:=fact*(f1-f2)/(2*gstep);
                      point[i]:=point[i]+gstep
```

```
                    end;
              slope:=norm(grad);
              for i:=1 to n do grad[i]:=grad[i]/slope
          end;
   procedure cfun;
          var
              x1,y1,phi:real;
*****************************************************************
 'out' --output procedure
*****************************************************************
          procedure out;
               var
                   jci,jcd:vector;
                   theta2,theta3,x,y,rot,drot,d:real;
                   i,j:integer;
               begin
                   cartes(jc,x1,y1,phi,theta2,theta3);
                   object^.x:=x1;object^.y:=y1;
                   object^.theta:=phi+jc[1]-1.5707963;
                   psobject^.x:=x1;psobject^.y:=y1;
                   psobject^.theta:=object^.theta;
                   x:=(x1-100.0)*gra;y:=(y1-100.0)*gra;
                   rot:=object^.theta*convert;
                   d:=(xg-x)*(xg-x)+(yg-y)*(yg-y);
                   drot:=(rot-rotg)*(rot-rotg);
                   if (d)28000) or (drot)=1000) then
                     begin
                         drawpolygon(object);
                         xg:=x;yg:=y;rotg:=rot
                     end
                end;
*****************************************************************
 'converge'--checks for convergence
*****************************************************************
           procedure converge;
                var
                    s1,s2:real;
                    i:integer;
                 begin
                    s1:=0.0;
                    for i:=1 to (n-1) do s1:=s1+(jc[i]-jcf[i])*(jc[i]-jcf[i])
                    s2:=(jc[3]-jcf[3])*(jc[3]-jcf[3]);
                    if (s1<=0.0005) then step:=step*factor;
                    if (slope<=slperr) or ((s1<=disterr)
                       and (s2<=disterr)) then goto 100
                 end;
            begin
                 out;
                 converge
            end;
*****************************************************************
 'Formobject' builds the polygon data structure
*****************************************************************
        procedure formobject;
            var
                x,y,xs,ys,x1,y1,a:real;
                i:integer;
                vertex,vpoint:pointer;
            begin
                vpoint:=nil;
```

```
          new(vertex);
          vertex1:=vertex;
          readln(afile,xs,ys);x:=xs;y:=ys;
          for i:=1 to nvert do
              begin
                  if (i<>1) then vertex^.l:=a;
                  vertex^.x:=x;vertex^.y:=y;
                  if (abs(x)>1.0e-3) then vertex^.theta:=arctan(y/x)
                  else vertex^.theta:=1.57079633;
                  if (x<0.0) then vertex^.theta:=vertex^.theta+3.1415927
                  else
                     if (y<0.0) then vertex^.theta:=vertex^.theta+6.2831654
                  vertex^.next:=vpoint;
                  if (i<>1) then vpoint^.prev:=vertex;
                  vpoint:=vertex;
                  if (i<>nvert) then
                    begin
                        readln(afile,x1,y1);
                        new(vertex)
                    end
                  else
                    begin
                        x1:=xs;y1:=ys
                    end;
                  a:=(x1-x)*(x1-x)+(y1-y)*(y1-y);
                  x:=x1;y:=y1
              end;
              vertex1^.next:=vpoint;vertex1^.l:=a;
              vpoint^.prev:=vertex1
        end;
{*****************************************************************************
 Psformobject' builds the data structure representing the hypothetical
 circles
*****************************************************************************}
   procedure psformobject;
        var
            x,y,rad,r:real;
            i:integer;
            vertex,vpoint:pointerp;
        begin
            vpoint:=nil;
            new(vertex);
            vertexp:=vertex;
            readln(afile,x,y,rad);
            for i:=1 to npseuds do
                begin
                    r:=sqrt(x*x+y*y);
                    vertex^.r:=r;
                    vertex^.rad:=rad;
                    if (abs(x)>1.0e-3) then vertex^.theta:=arctan(y/x)
                    else vertex^.theta:=1.57079633;
                    if (x<0.0) then vertex^.theta:=vertex^.theta+3.1415927
                    else
                       if (y<0.0) then vertex^.theta:=vertex^.theta+6.2831654
                    vertex^.next:=vpoint;
                    vpoint:=vertex;
                    if (i<>npseuds) then
                      begin
                          readln(afile,x1,y1,rad);
                          new(vertex)
                      end
```

```
                        end
                   end;
                   vertexp .next:=vpoint
               end;
   begin  {*********main program***************}
**************************************************************************
  y0 -height of the plane of movement above the global plane z=0
  onum -number of obstacles
  nvert--number of vertices of the polygonal object
  (xi,yi,iphi) and (xf,yf,fphi)-- initial and final positions of the object
  l0,l1,l2,l3---lengths of the links of the manipulator
  npseuos--number of hypothetical circles
  llim,ulim--lower and upper limits of the joint angles
  step---move step,gstep--step used to compute the gradient
  c0,c1,c2--parameters of the objective function
**************************************************************************
       start;
       connect(afile,'inouto','dat','r',p);
       reset(afile);
       readln(afile,y0);
       readln(afile,onum);
       readln(afile,nvert);
       readln(afile,xi,yi,iphi);
       readln(afile,xf,yf,fphi);
       formobject;
       new(object);
       object^.next:=vertexi;
       readln(afile,npseuds);
       psformobject;
       new(psobject);
       psobject .x:=xi;psobject .y:=yi;psobject^.theta:=iphi;
       psobject^.next:=vertexp;
       for i:=0 to 4 do
          begin
               readln(afile,llim[i],ulim[i]);
               fact[i]:=ulim[i]-llim[i]
          end;
       readln(afile,l0,l1,l2,l3);
       jointspace(xi,yi,iphi,jc);
       jointspace(xf,yf,fphi,jcf);
       xg:=(xi-100.0)*gra;yg:=(yi-100.0)*gra;rotg:=iphi*convert;
**********************************************************************
 obstacles being formed
**********************************************************************
       for i:=1 to onum do
          begin
               readln(afile,x1,y1,phi,nvert);
               new(obstacle[i]);
               obstacle[i]^.x:=x1;obstacle[i]^.y:=y1;
               obstacle[i]^.theta:=phi;
               formobject;
               obstacle[i]^.next:=vertex1;
               getinfo(obstacle[i],vertex1,x1,y1);
               ox:=trunc((x1-100.0)*gra);oy:=trunc((y1-100.0)*gra);
               beginp(ox,oy,1);
               fill(6);
               drawpolygon(obstacle[i]);
               endp
          end;
        object^.x:=xf;object^.y:=yf;object^.theta:=fphi;
```

```
drawpolygon(object);
object^.x:=x1;object^.y:=y1;object^.theta:=iphi;
drawpolygon(object);
readln(afile, c0, c1, c2);
readln(afile, step, gstep, factor, rotfact1, rotfact2, rotfact3, slperr, disterr)
readln(afile, approach, dist1, dist2);
clstobstacle:=[];check:=true;count:=0;
gradient(jc, grad, slope);
for i:=1 to n do jc[i]:=jc[i]-step*grad[i];
dfp;
for i:=1 to n do s[i]:=-grad[i];
repeat
      for i:=1 to n do jc1[i]:=jc[i]+step*s[i];
      gradient(jc1, grad, slope);
      for i:=1 to n do s[i]:=s[i]-grad[i];
      s1:=norm(s);
      for i:=1 to n do
          begin
             s[i]:=s[i]/s1;
             jc[i]:=jc[i]+step*s[i]
          end;
      count:=count+1;
      if ((count mod nstep)=0) then  check:=true  else check:=false;
      dfp
until (slope<=slperr);
100:
end.
```

APPENDIX B

PROGRAM IMPLEMENTING THE CONFIGURATION SPACE ALGORITHM

```
program avoidobstacle(input,output);
  const
*************************************************************************
  convert--conversion factor from radians to degrees
  maxct,minct--arbitrary large and small constants
*************************************************************************
      convert=57.29577951;
      gra=68.2667;
      maxct=1.0e+5;minct=5.0e-4;
      pi=3.1415927;
      step=0.5;
  type
*************************************************************************
  pointer--node representing a vertex of a polygon
  polygon--header of the structure representing a polygon
  stree--node of the path list
  gpointer--node representing the vertices of the 3-d expanded obstacle
  gpolygon--header for the 3-d obstacle
*************************************************************************
      pointer=^node;
      node=record
                 x,y:real;
                 next,prev:pointer
              end;
      polygon=^lframe;
      lframe=record
                 x,y,theta:real;
                 next:pointer
             end;
      stree=^treept;
      treept=record
                  x,y,phi:real;
                  son,parent:stree
             end;
      gpointer=^gnode;
      gnode=record
                 x,y:array [-50..50] of real;
                 next,prev:gpointer
            end;
      gpoly=^glframe;
      glframe=record
                   x,y,theta:real;
                   next:gpointer
              end;
  var
*************************************************************************
  gobstacle--array of obstacles
*************************************************************************
      phi,margin,xmax,xmin,ymax,ymin,thetamax,thetamin,xf,yf,fphi,
      xi,yi,iphi:real;
      dtheta,xo,yo,factor:real;
      tslices,ox,oy,i,k,onum,p,nvert,nvobj,ndraw:integer;
      vertex1:pointer;
      gobstacle:array [1..10] of gpoly;
      gvertex1:gpointer;
      object,obstacle:polygon;
      afile:text;
      root,leaf:stree;
      feasible:boolean;
$include (pas)ndsup:pas
```

```
{$include (pas)segment:pas
*****************************************************************************
  'free' disposes off the nodes of a circular list that is not needed
*****************************************************************************
  procedure free(vertex:pointer);
         var
             vertex1:pointer;
         begin
             vertex^.prev^.next:=nil;
             while (vertex^.next<>nil) do
               begin
                   vertex1:=vertex^.next;
                   dispose(vertex);
                   vertex:=vertex1
               end;
             dispose(vertex)
         end;
*****************************************************************************
  'freetree' disposes off the redundant nodes of the pathlist
*****************************************************************************
  procedure freetree(node:stree);
       var
          node1,node2:stree;
       begin
           node1:=node^.son;
           while (node1<>nil) do
              begin
                   node2:=node1^.son;
                   dispose(node1);
                   node1:=node2
              end;
            node^.son:=nil
       end;
*****************************************************************************
  'getinfo' computes the global coord.'(x,y)' of the vertex 'vertex'
   belonging to the polygon 'object'
*****************************************************************************
  procedure getinfo(object:polygon;vertex:pointer;var x,y:real);
       var
            rad,phi,x1,y1,x2,y2:real;
       begin
            x1:=object^.x;y1:=object^.y;
            x2:=vertex^.x;y2:=vertex^.y;
            rad:=sqrt(x2*x2+y2*y2);
            if (abs(x2)<=minct) then
              if (y2<0.0) then phi:=3.0*pi/2.0 else phi:=pi/2.0
            else
              begin
                   phi:=arctan(y2/x2);
                   if (x2<0.0) then phi:=phi+pi
              end;
            phi:=phi+object^.theta;
            x:=x1+rad*cos(phi);
            y:=y1+rad*sin(phi)
       end;
*****************************************************************************
  'linetopt' computes the distance of the point '(xc,yc)' from the line
   ing the points '(xa,ya)' and '(xb,yb)'.The distance is given an arbit
   large value 'maxct' if the perp. from the pt. to the line does'nt fal
   within the segment
```

```
**********************************************************************
  function linetopt(xa,ya,xb,yb,xc,yc:real):real;
        var
            l,d1,d2,cos1,cos2,a,b:real;
        begin
            l:=(xa-xb)*(xa-xb)+(ya-yb)*(ya-yb);
            d1:=(xa-xc)*(xa-xc)+(ya-yc)*(ya-yc);
            d2:=(xb-xc)*(xb-xc)+(yb-yc)*(yb-yc);
            if (abs(d1)<minct) or (abs(d2)<minct) then linetopt:=0.0
            else
              begin
                   cos1:=d1+l-d2;cos2:=d2+l-d1;
                   a:=2.0*sqrt(d1*l);b:=2.0*sqrt(d2*l);
                   if (abs(1.0-cos1/a)<minct) or (abs(1.0+cos1/a)<minct) then
                     linetopt:=0.0
                   else
                     begin
                         if (abs(cos1/a)<minct) then linetopt:=sqrt(d1)
                         else if (abs(cos2/b)<minct) then linetopt:=sqrt(d2)
                         else
                           if ((cos1>0.0) and (cos2>0.0)) then
                             linetopt:=sqrt(d2-cos2*cos2/(4*l))
                           else linetopt:=maxct
                     end
              end
        end;
*******************************************************************
  'max' checks if 't' is large(maxct) or not
*******************************************************************
function max(t:real):boolean;
   begin
       if (abs(t-maxct)<100.0) then max:=true
       else max:=false
   end;
******************************************************************
  'arccos' computes the cosine inverse of 'cx'
******************************************************************
function arccos(cx:real):real;
   var
       sx,a:real;
   begin
       if (abs(cx-1.0)<minct) then arccos:=0.0
         else
           if (abs(cx+1.0)<minct) then arccos:=pi
           else
             begin
                 sx:=sqrt(1-cx*cx);
                 if (abs(cx)>minct) then
                    begin
                        a:=arctan(sx/cx);
                        if (cx<0.0) then a:=a+pi
                    end
                 else a:=pi/2.0;
                 arccos:=a
             end
   end;
**********************************************************************
  'formobject'  forms the data structure representing a polygon whose
   position is given by '(xo,yo,phi)'
**********************************************************************
```

```
procedure formobject(xo,yo,phi:real);
    var
        x,y,r,zi:real;
        i:integer;
        vertex,vpoint:pointer;
    begin
        vpoint:=nil;
        new(vertex);
        vertex1:=vertex;
        readln(afile,x,y);
        for i:=1 to nvert do
            begin
                if (abs(phi)<minct) then
                  begin
                      vertex^.x:=xo+x;vertex^.y:=yo+y
                  end
                else
                  begin
                      r:=sqrt(sqr(x)+sqr(y));
                      if (abs(x)<minct) then
                        if (y<0.0) then zi:=3.0*pi/2.0 else zi:=pi/2.0
                      else
                        begin
                            zi:=arctan(y/x);
                            if (x<0.0) then zi:=zi+pi
                        end;
                      zi:=zi+phi;
                      vertex^.x:=xo+r*cos(zi);
                      vertex^.y:=yo+r*sin(zi)
                  end;
                vertex^.next:=vpoint;
                if (i<>1) then vpoint^.prev:=vertex;
                vpoint:=vertex;
                if (i<>nvert) then
                  begin
                      readln(afile,x,y);
                      new(vertex);
                  end
            end;
            vertex1^.next:=vpoint;
            vpoint^.prev:=vertex1
        end;
*****************************************************************************
  'psformobject' expands the polygon 'object' by 'margin' on all sides
*****************************************************************************
  procedure psformobject(object:polygon;margin:real);
      var
          o,phi,r1,r2,r3,x1,y1,x2,y2,x3,y3,x4,y4,rado:real;
          vertex,vpoint,pvertex:pointer;
          enter:boolean;
      begin
          vertex:=object^.next;enter:=false;
          new(pvertex);
          vertex1:=pvertex;
          vpoint:=nil;
          repeat
              x1:=vertex^.x;y1:=vertex^.y;
              x2:=vertex^.next^.x;y2:=vertex^.next^.y;
              x3:=vertex^.prev^.x;y3:=vertex^.prev^.y;
              r1:=(x1-x2)*(x1-x2)+(y1-y2)*(y1-y2);
```

```
                    r2:=(x1-x3)*(x1-x3)+(y1-y3)*(y1-y3);
                    r3:=(x2-x3)*(x2-x3)+(y2-y3)*(y2-y3);
                    phi:=arccos((r1+r2-r3)/(2.0*sqrt(r1*r2)));
                    c:=1/cos(phi/2.0);
                    phi:=(pi-phi)/2.0;
                    radd:=margin/cos(phi);
                    x2:=x1+c*(x2-x1)/sqrt(r1);y2:=y1+c*(y2-y1)/sqrt(r1);
                    x3:=x1+c*(x3-x1)/sqrt(r2);y3:=y1+c*(y3-y1)/sqrt(r2);
                    x4:=(x3+x2)/2.0;y4:=(y3+y2)/2.0;
                    pvertex^.x:=x1+radd*(x1-x4);
                    pvertex^.y:=y1+radd*(y1-y4);
                    pvertex^.next:=vpoint;
                    if (enter) then vpoint^.prev:=pvertex;
                    vpoint:=pvertex;
                    enter:=true;
                    vertex:=vertex^.prev;
                    if (vertex<>object^.next) then  new(pvertex)
                until (vertex=object^.next);
                vertex1^.next:=vpoint;vpoint^.prev:=vertex1
        end;
    procedure drawcircle(x,y,r:real);
        var
            x1,y1:integer;
            theta,dtheta:real;
        begin
            theta:=0.0;dtheta:=0.2;
            y1:=trunc(y);
            x1:=trunc(x+r);
            move(x1,y1);
            repeat
                theta:=theta+dtheta;
                x1:=trunc(x+r*cos(theta));
                y1:=trunc(y+r*sin(theta));
                draw(x1,y1)
            until (theta)=6.2831854)
        end;
    procedure drawpolygon(obj:polygon);
        var
            theta,x,y:real;
            x1,y1:integer;
            vertex:pointer;
        begin
            vertex:=obj^.next;
            if (obj=object) then getinfo(obj,vertex,x,y)
            else
              begin
                  x:=vertex^.x;y:=vertex^.y
              end;
            x1:=trunc((x-100.0)*gra);y1:=trunc((y-100.0)*gra);
            if (x1<0) then x1:=0;if (y1<0) then y1:=0;
            if (x1>4090) then x1:=4090;if (y1>3080) then y1:=3080;
            move(x1,y1);
            repeat
                  vertex:=vertex^.next;
                  if (obj=object) then getinfo(obj,vertex,x,y)
                  else
                    begin
                        x:=vertex^.x;y:=vertex^.y
                    end;
                  x1:=trunc((x-100.0)*gra);y1:=trunc((y-100.0)*gra);
```

```
            if (x1<0) then x1:=0;if (y1<0) then y1:=0;
            if (x1>4090) then x1:=4090;if (y1>3080) then y1:=3080;
            draw(x1,y1)
        until (vertex=obj^.next)
    end;
procedure traceobject(k,l:integer);
    var
        vertex:gpointer;
        x,y:real;
    begin
        vertex:=gobstacle[l]^.next;
        object^.theta:=k*dtheta;
        repeat
            x:=vertex^.x[k];y:=vertex^.y[k];
            object^.x:=x;object^.y:=y;
            drawpolygon(object);
            vertex:=vertex^.next
        until (vertex=gobstacle[l]^.next)
    end;
*************************************************************************
  'wall' checks if the point '(x,y)' is within the specified workspace
*************************************************************************
  function wall(x,y:real):boolean;
      begin
          if (x>=xmax) or (y>=ymax) then wall:=true
          else
              if (x<=xmin) or (y<=ymin) then wall:=true
              else wall:=false
      end;
  procedure trace(node1,node2:stree);
      var
          xa,xb,ya,yb,d,dx,dy,bphi,aphi,dphi:real;
          i,ndraw:integer;
      begin
          xa:=node1^.x;ya:=node1^.y;aphi:=node1^.phi;
          xb:=node2^.x;yb:=node2^.y;bphi:=node2^.phi;
          d:=sqrt(sqr(xb-xa)+sqr(yb-ya)+sqr(bphi-aphi));
          dx:=(xb-xa)/d;dy:=(yb-ya)/d;dphi:=(bphi-aphi)/d;
          ndraw:=trunc(d/1.75);
          for i:=1 to ndraw do
              begin
                  xa:=xa+1.75*dx;ya:=ya+1.75*dy;aphi:=aphi+1.75*dphi;
                  object^.x:=xa;object^.y:=ya;object^.theta:=aphi;
                  drawpolygon(object)
              end
      end;
  procedure abort;
      begin
          writeln('Search aborted.Path does not exist.');
          goto 100
      end;
*************************************************************************
  'ponkint' determines the points of intersection of the line joining the
  points '(xa,ya)' and '(xb,yb)' with the 'k''th slice of the 3-d
  obstacle 'gobs'.The pts. are specified in terms of 't1' and 't2' where
  ti=0 is (xa,ya) and ti=1 is (xb,yb).
*************************************************************************
 procedure ponkint(xa,ya,xb,yb:real;gobs:gpoly;k:integer;var t1,t2:real)
      var
          a,b,c,d,xc,yc,xd,yd,t,den:real;
```

```
              if (x1<0) then x1:=0;if (y1<0) then y1:=0;
              if (x1>4090) then x1:=4090;if (y1>3080) then y1:=3080;
              draw(x1,y1)
          until (vertex=obj^.next)
      end;
procedure traceobject(k,l:integer);
      var
          vertex:gpointer;
          x,y:real;
      begin
          vertex:=gobstacle[l]^.next;
          object^.theta:=k*dtheta;
          repeat
               x:=vertex^.x[k];y:=vertex^.y[k];
               object^.x:=x;object^.y:=y;
               drawpolygon(object);
               vertex:=vertex^.next
          until (vertex=gobstacle[l]^.next)
      end;
*****************************************************************************
  'wall' checks if the point '(x,y)' is within the specified workspace
*****************************************************************************
  function wall(x,y:real):boolean;
       begin
           if (x>=xmax) or (y>=ymax) then wall:=true
           else
              if (x<=xmin) or (y<=ymin) then wall:=true
              else wall:=false
       end;
  procedure trace(node1,node2:stree);
       var
           xa,xb,ya,yb,d,dx,dy,dphi,aphi,bphi:real;
           i,ndraw:integer;
       begin
           xa:=node1^.x;ya:=node1^.y;aphi:=node1^.phi;
           xb:=node2^.x;yb:=node2^.y;bphi:=node2^.phi;
           d:=sqrt(sqr(xb-xa)+sqr(yb-ya)+sqr(bphi-aphi));
           dx:=(xb-xa)/d;dy:=(yb-ya)/d;dphi:=(bphi-aphi)/d;
           ndraw:=trunc(d/1.75);
           for i:=1 to ndraw do
               begin
                   xa:=xa+1.75*dx;ya:=ya+1.75*dy;aphi:=aphi+1.75*dphi;
                   object^.x:=xa;object^.y:=ya;object^.theta:=aphi;
                   drawpolygon(object)
               end
        end;
   procedure abort;
        begin
            writeln('Search aborted.Path does not exist.');
            goto 100
        end;
*****************************************************************************
  'pchkint' determines the points of intersection of the line joining the
  points '(xa,ya)' and '(xb,yb)' with the 'k''th slice of the 3-d
  obstacle 'gobs'.The pts. are specified in terms of 't1' and 't2' where
  ti=0 is (xa,ya) and ti=1 is (xb,yb).
*****************************************************************************
  procedure pchkint(xa,ya,xb,yb:real;gobs:gpoly;k:integer;var t1,t2:real)
       var
           a,b,c,d,xc,yc,xd,yd,t,den:real;
```

```
           yah:boolean;
         begin
             t1:=maxct;t2:=0.0-maxct;
             if (abs(aphi-bphi)<=0.5*dtheta) then
               begin
                   k:=getk(aphi);
                   pchkint(xa,ya,xb,yb,gobs,k,t1,t2);
               end
             else
               begin
                   j:=0;
                   d:=sqrt(sqr(xb-xa)+sqr(yb-ya)+sqr(aphi-bphi));
                   dx:=(xb-xa)/d;dy:=(yb-ya)/d;dphi:=(bphi-aphi)/d;
                   t:=step;
                   repeat
                        x:=xa+t*dx;y:=ya+t*dy;phi:=aphi+t*dphi;
                        k:=getk(phi);
                        ptinpoly(x,y,k,gobs,yah);
                        if (yah) then
                          begin
                               d1:=(t-step/2.0)/d:
                               if (d1<t1) then t1:=d1;
                               if (d1>t2) then t2:=d1
                          end;
                        t:=t+step
                   until (t>=d);
                   if max(-t2) then t2:=maxct
               end
         end;
*************************************************************************
  'append' adds to the pathlist,any new feasible point on the path that is
  obtained
*************************************************************************
procedure append(anode:stree;xb,yb,bphi:real;var treenode:stree);
      begin
          new(treenode);
          treenode^.x:=xb;treenode^.y:=yb;treenode^.phi:=bphi;
          treenode^.son:=nil;
          anode^.son:=treenode;
          treenode^.parent:=anode
      end;
*************************************************************************
  'repetition' checks if the new point '(x,y)' obtained is a repetition of
  an earlier state of the object.'node' is the last node of the pathlist
*************************************************************************
function repetition(x,y:real;node:stree):boolean;
      var
          x1,y1,x2,y2,d:real;
          term:boolean;
      begin
          x1:=node^.x;y1:=node^.y;term:=false;
          while (node^.parent<>nil) and (not(term)) do
               begin
                    x2:=node^.parent^.x;y2:=node^.parent^.y;
                    d:=linetopt(x1,y1,x2,y2,x,y);
                    if (d<=margin) then term:=true
                    else
                      begin
                           x1:=x2;y1:=y2;
                           node:=node^.parent
```

```
                    end
                end;
            repetition:=term;
        end;
procedure retrace;
        var
            tnode:stree;
        begin
            tnode:=root;
            while (tnode^.son<>nil) do
                begin
                    trace(tnode,tnode^.son);
                    tnode:=tnode^.son
                end
        end;
{**************************************************************************
  'modifytree' deletes all the nodes between 'node1' and 'node2' in the
   'pathlist'
**************************************************************************}
procedure modifytree(node1,node2:stree);
        begin
            node2^.parent^.son:=nil;
            freetree(node1);
            node1^.son:=node2;node2^.parent:=node1
        end;
{**************************************************************************
  'reduce' refines the path from  the point given by 'anode' of the pathlist
  to that given by 'bnode','cnode' being the intermediate point.
**************************************************************************}
procedure reduce(anode,cnode,bnode:stree);
        var
            node1,node2:stree;
            yes:boolean;
            xa,ya,aphi,xb,yb,bphi:real;
        procedure chksafe(xa,ya,aphi,xb,yb,bphi:real);
            var
                i:integer;
                t1,t2:real;
            begin
                i:=1;yes:=true;
                repeat
                    gchkint(xa,ya,aphi,xb,yb,bphi,gobstacle[i],t1,t2);
                    if (t1<1.0) then yes:=false;
                    i:=i+1
                until (i=onum+1) or not(yes)
            end;
        begin
            node1:=anode;
            repeat
                if (node1=anode) then node2:=bnode^.parent else node2:=bnode;
                xa:=node1^.x;ya:=node1^.y;aphi:=node1^.phi;
                repeat
                    xb:=node2^.x;yb:=node2^.y;bphi:=node2^.phi;
                    chksafe(xa;ya,aphi,xb,yb,bphi);
                    if (yes) then modifytree(node1,node2)
                    else node2:=node2^.parent
                until (node2=cnode) or (yes);
                node1:=node1^.son
            until (node1=cnode) or (yes)
        end;
```

```
{*******************************************************************************
  'auxfindpath' searches the line joining points '(xa,ya)' and '(xb,yb)'
  on the plane of the 'k''th slices for the intermediate points on this plane.
*******************************************************************************}
procedure auxfindpath(anode:stree;xa,ya,xb,yb:real;k,l:integer;var ta,tb
            var
                 xc,yc,xd,yd,t1,t2,da,db:real;
                 i:integer;
            begin
                 da:=maxct;db:=maxct;
                 pchkint(xa,ya,xb,yb,gobstacle[l],k,ta,tb);
                 for i:=1 to onum do
                    if (i<>l) then
                      begin
                           pchkint(xa,ya,xb,yb,gobstacle[i],k,t1,t2);
                           if not(max(t1)) and not(max(t2)) then
                             begin
                                  if (t1<ta) and (t2>tb) then
                                    begin
                                         ta:=t1;tb:=t2
                                    end;
                                  if (t1<ta) and (t2<tb) and (t2>ta) then ta:=t1;
                                  if (t1>ta) and (t1<tb) and (t2>tb) then tb:=t2
                                  if (t2<ta) then
                                    if (ta-t2<da) then da:=ta-t2;
                                  if (t1>tb) then
                                    if (t1-tb<db) then db:=t1-tb
                             end
                      end;
                 if not(max(da)) then ta:=ta-da/2.0 else ta:=ta-5.0*margin;
                 if not(max(db)) then tb:=tb+db/2.0 else tb:=tb+5.0*margin;
                 xc:=xa+ta*(xb-xa);yc:=ya+ta*(yb-ya);
                 xd:=xa+tb*(xb-xa);yd:=ya+tb*(yb-ya);
                 if (wall(xc,yc)) or (repetition(xc,yc,anode)) then ta:
                 if (wall(xd,yd)) or (repetition(xd,yd,anode)) then tb:
        end;
{*******************************************************************************
  'fp' is the findpath procedure  that calls itself recursively.'an' is the
  last node in the pathlist and '(xb,yb,bphi)' gives the position of the
  object to which a path is to be found.'(xr,yr,rphi)' is the position of
  the midpt. of intersection with the closest obstacle in the earlier
  recursion step.'w' tells the side of the normal plane through (xr,yr,rphi)
  on which the intermediate pt. lies.
*******************************************************************************}
  procedure fp(an:stree;xb,yb,bphi,xr,yr,rphi:real;w:integer;var bn:stree;
               var fea:boolean);
        var
             aphi,pphi,xa,ya,xp,yp,t1,t2,t,dist:real;
             i,k:integer;
             sea:boolean;
        procedure pathfind;
             type
{*******************************************************************************
  'pt'-node representing an intermediate point
*******************************************************************************}
                     pt=^pts;
                     pts=record
                               x,y,t,phi:real;
                               next:pt
                         end;
```

```
            var
                 dx,dy,dphi,phi,a,b,c,x,y,x1,y1,d,xs,ys,sphi,t,xm,ym,
                 ta,tb:real;
                 i,j,n,nnode:integer;
                 head:pt;
                 yes:boolean;
***************************************************************************
  'free2' disposes off all the nodes in the list of intermediate pts. to the
  right of 'node'.
  'appendlist' adds another node to this list in the order of increasing 'tc'.
***************************************************************************
            procedure free2(node:pt);
                 var
                      node1:pt;
                 begin
                      while (node<>nil) do
                           begin
                                node1:=node^.next;
                                dispose(node);
                                node:=node1
                           end
                 end;
            procedure appendlist(xc,yc,phi,tc:real);
                 var
                      node,tnode1,tnode2:pt;
                      term:boolean;
                 begin
                      new(node);
                      node^.x:=xc;node^.y:=yc;node^.phi:=phi;node^.t:=tc;
                      if (head=nil) then
                        begin
                             head:=node;head^.next:=nil
                        end
                      else
                        if (tc<head^.t) then
                          begin
                               node^.next:=head;head:=node
                          end
                        else
                          begin
                               tnode1:=head;term:=false:
                               while (tnode1^.next<>nil) and (not(term)) do
                                     begin
                                          tnode2:=tnode1^.next;
                                          if (tc<tnode2^.t) then
                                            begin
                                                 tnode1^.next:=node;
                                                 node^.next:=tnode2;
                                                 term:=true
                                            end
                                          else tnode1:=tnode2
                                     end;
                               if (tnode1^.next=nil) then
                                 begin
                                      tnode1^.next:=node;node^.next:=nil
                                 end
                          end;
                        if (nnode>10) then
                          begin
                               tnode1:=head;
```

```
                    while (tnode1^.next^.next<>nil) do
                       tnode1:=tnode1^.next;
                    dispose(tnode1^.next);tnode1^.next:=nil
                end
      end;
procedure chkpath(x,y,phi:real);
      var
          cn:stree;
      begin
          fp(an,x,y,phi,xb,yb,bphi,1,cn,fea);
          if (fea) then
            begin
                fp(cn,xb,yb,bphi,xo,yo,pphi,2,bn,fea);
                if (fea) then reduce(an,cn,bn)
            end
      end;
procedure fcheck(x1,y1:real;var t:real);
      var
          x,y,d:real;
      begin
          if not(max(t)) then
            begin
                x:=xm+t*x1;y:=ym+t*y1;
                d:=dx*(x-xr)+dy*(y-yr)+dphi*(phi-rphi);
                if (d<0.0) then t:=maxct
            end
      end;
procedure recursion;
      var
         xc,yc,cphi:real;
         node,node1:pt;
      begin
          node:=head;
          if (head<>nil) then
            repeat
                 xc:=node^.x;yc:=node^.y;
                 cphi:=node^.phi;
                 chkpath(xc,yc,cphi);
                 if (not(fea)) then
                   begin
                       node1:=node^.next;
                       dispose(node);node:=node1;
                       freetree(an)
                   end
            until (fea) or (node=nil);
          free2(node);
          if (node=nil) or (head=nil) then
            begin
                bn:=nil;fea:=false
            end
      end;
procedure formlist(x1,y1,t:real;i:integer);
      var
          xn,yn,ta,tb,xc,yc,tc,phi:real;
      begin
          phi:=i*dtheta;
          xn:=xm+x1;yn:=ym+y1;
          auxfindpath(an,xm,ym,xn,yn,i,k,ta,tb);
          if (w<>0) then
            begin
```

```
                fcheck(x1,y1,ta);fcheck(x1,y1,tb)
              end;
            if (not(max(ta))) then
              begin
                nnode:=nnode+1;
                tc:=sqrt(factor*sqr(t)+sqr(ta));
                xc:=xm+ta*x1;yc:=ym+ta*y1;
                appendlist(xc,yc,phi,tc)
              end;
            if (not(max(tb))) then
              begin
                nnode:=nnode+1;
                tc:=sqrt(factor*sqr(t)+sqr(tb));
                xc:=xm+tb*x1;yc:=ym+tb*y1;
                appendlist(xc,yc,phi,tc)
              end
       end;
begin  {*** pathfind ***}
     head:=nil;nnode:=0;
     if (w<>0) then
       if (w=1) then
         begin
           dx:=xa-xr;dy:=ya-yr;dphi:=aphi-rphi;
         end
       else
         begin
           dx:=xb-xr;dy:=yb-yr;dphi:=bphi-rphi;
         end;
     a:=xb-xa;b:=yb-ya;
     c:=sqrt(sqr(a)+sqr(b)+sqr(bphi-aphi));
     if (abs(a/c)<1.0e-3) and (abs(b/c)<1.0e-3) then
       begin
           x1:=0.0;i:=getk(pphi);
           repeat
                y1:=sqrt(1.0-sqr(x1));
                xm:=xp;ym:=yp;t:=0.0;
                formlist(x1,y1,0.0,i);formlist(-y1,x1,0.0,i);
                x1:=x1+0.1
           until (x1>1.0);
           recursion
       end
     else
       begin
           if (abs(b)<minct) then
             begin
                x1:=0.0;y1:=1.0
             end
           else
             if (abs(a)<minct) then
               begin
                  y1:=0.0;x1:=1.0
               end
             else
               begin
                  x1:=1.0;y1:=-a/b
               end;
           d:=sqrt(sqr(x1)+sqr(y1));
           x1:=x1/d;y1:=y1/d;
           xs:=y1*(bphi-aphi);ys:=-x1*(bphi-aphi);
           sphi:=x1*b-y1*a;
```

```
                d:=sqrt(sqr(xs)+sqr(ys)+sqr(sphi));
                xs:=xs/d;ys:=ys/d;sphi:=sphi/d;
                if not(sea) then
                  begin
                       xm:=xp;ym:=yp;i:=getk(pphi);
                       formlist(x1,y1,0.0,i);
                       recursion
                  end
                else
                  begin
                       n:=k;
                       for i:=-tslices to tslices do
                          begin
                               k:=n;
                               phi:=i*dtheta;
                               t:=(phi-pphi)/sphi;
                               xm:=xp+t*xs;ym:=yp+t*ys;
                               ptinpoly(xm,ym,i,gobstacle[k],yes);
                               if not(yes) then
                                 begin
                                      k:=0;j:=1;
                                      while (k=0) and (j<=onum) do
                                            begin
                                                 if (j<>n) then
                                                   begin
                                                        ptinpoly(xm,ym
                                                           gobstacle[j],yes);
                                                        if (yes) then
                                                   end;
                                                 j:=j+1
                                            end
                                 end;
                               if (k<>0) then
                                 begin
                                      x:=0.0;
                                      repeat
                                            y:=sqrt(1.0-sqr(x));
                                            formlist(x,y,t,i);
                                            formlist(-y,x,t,i);
                                            x:=x+0.25
                                      until (x>1.0)
                                 end
                          end;
                        recursion
                  end
           end
      end;
begin  {***** findpath ****}
    xa:=an^.x;ya:=an^.y;aphi:=an^.phi;
    dist:=sqrt(sqr(xb-xa)+sqr(yb-ya)+sqr(bphi-aphi));
    t:=1.0;k:=0;
    for i:=1 to onum do
       begin
           gchkint(xa,ya,aphi,xb,yb,bphi,gobstacle[i],t1,t2);
           if (t1<t) then
             begin
                  if ((t2-t1)*dist<margin/2.0) then sea:=false
                    else sea:=true;
                  t:=t1;k:=i;
                  xp:=xa+(t1+t2)*(xb-xa)/2.0;yp:=ya+(t1+t2)*(yb-ya)/2.0;
```

```
                    pphi:=aphi+(t1+t2)*(ophi-aphi)/2.0
                end
            end;
        if (k=0) then
          begin
            append(an,xb,yb,bphi,bn);
            object^.x:=xb;object^.y:=yb;
            object^.theta:=bphi;
            drawpolygon(object);
            fea:=true
          end
        else pathfind
    end;
****************************************************************************
  'gformobstacle' forms the data structure representing the 3-d obstacles'
****************************************************************************
   procedure gformobstacle;
        var
           n,i:integer;
           vertex,vpoint:gpointer;
        begin
           n:=nvert+nvobj;
           vpoint:=nil;
           new(vertex);
           gvertex1:=vertex;
           for i:=1 to n do
              begin
                 vertex^.next:=vpoint;
                 if (i<>1) then vpoint^.prev:=vertex;
                 vpoint:=vertex;
                 if (i<>n) then new(vertex)
              end;
              gvertex1^.next:=vpoint;
              vpoint^.prev:=gvertex1
        end;
****************************************************************************
  'appendobstacle' forms the envelope for obstacle 'i',the orientation of the
  object being given by 'k'
****************************************************************************
  procedure appendobstacle(i,k:integer);
       var
          theta,xa,ya,xb,yb,xc,yc,xd,yd,x,y,a,b,c,d,d1,cos1,cos2,x0,y0,
          phi1,phi2:real;
          vertexa,vertexb,vertex:pointer;
          gvertex:gpointer;
       begin
          d1:=0.0;theta:=dtheta*k;x0:=obstacle^.x;y0:=obstacle^.y;
          object^.x:=0.0;object^.y:=0.0;object^.theta:=theta;
          vertexa:=obstacle^.next;
          xa:=vertexa^.x;ya:=vertexa^.y;
          xb:=vertexa^.prev^.x;yb:=vertexa^.prev^.y;
          a:=yb-ya;b:=xa-xb;
          c:=a*x0+b*y0+xb*ya-xa*yb;
          vertex:=object^.next;
          repeat
               getinfo(object,vertex,x,y);xd:=xa-x;yd:=ya-y;
               d:=a*xd+b*yd+xb*ya-xa*yb;
               if ((c<0.0) and (d>=0.0)) or ((c>0.0) and (d<=0.0)) then
                 begin
                     d:=abs((a*xd+b*yd+xb*ya-xa*yb)/sqrt(a*a+b*b));
```

```
                    if (d)=d1) then
                       begin
                            xc:=-x;yc:=-y;
                            d1:=d;vertexb:=vertex
                       end
                  end;
                vertex:=vertex^.prev
          until (vertex=object^.next);
          object^.next:=vertexb;
          gvertex:=gobstacle[1]^.next;
          x:=xc;y:=yc;
          repeat
                xc:=xa+x;yc:=ya+y;
                gvertex^.x[k]:=xc;gvertex^.y[k]:=yc;
                object^.x:=xc;object^.y:=yc;
                getinfo(object,vertexb^.prev,xb,yb);
                getinfo(object,vertexb^.next,xc,yc);
                xd:=vertexa^.prev^.x;yd:=vertexa^.prev^.y;
                a:=sqr(xb-xa)+sqr(yb-ya);
                b:=sqr(xc-xa)+sqr(yc-ya);
                c:=sqr(xd-xa)+sqr(yd-ya);
                d:=sqr(xc-xb)+sqr(yc-yb);
                d1:=sqr(xd-xc)+sqr(yd-yc);
                cos1:=(a+b-d)/(2.0*sqrt(a*b));cos2:=(b+c-d1)/(2.0*sqrt(b*c));
                phi1:=arccos(cos1);phi2:=arccos(cos2);
                a:=phi1+phi2;
                if (a)=pi) then
                  begin
                       vertexb:=vertexb^.prev;
                       object^.x:=0.0;object^.y:=0.0;
                       getinfo(object,vertexb,x,y);
                       x:=-x;y:=-y
                  end
                else
                    begin
                         vertexa:=vertexa^.prev;
                         xa:=vertexa^.x;ya:=vertexa^.y
                    end;
                gvertex:=gvertex^.prev
            until (gvertex=gobstacle[1]^.next)
      end;
  begin  {**********main program**************}
************************************************************************
  onum--number of obstacles
  (xi,yi,iphi)--initial position
  (xf,yf,fphi)--final position
  nvobj--no. of vertices of the object
  xmin,xmax--x limits,ymin,ymax--y limits,thetamin,thetamax--theta limits
  margin--safety margin
  tslices--no. of slices
  dtheta--orientation increment
  root,leaf--first and last nodes of the pathlist
************************************************************************
      start;
      connect(afile,'input1-new','dat','r',p);
      reset(afile);
      readln(afile,onum);
      readln(afile,xi,yi,iphi);
      readln(afile,xf,yf,fphi);
      iphi:=iphi/convert;fphi:=fphi/convert;
```

```
      readln(afile,nvobj);
      nvert:=nvobj;
      formobject(0.0,0.0,0.0);
      new(object);
      object  .next:=vertex1;
      readln(afile,xmin,xmax,ymin,ymax,thetamin,thetamax);
      readln(afile,margin);
      readln(afile,tslices);
      thetamin:=thetamin/convert;thetamax:=thetamax/convert;
      dtheta:=(thetamax-thetamin)/tslices;
      tslices:=trunc(tslices/2);
      readln(afile,ndraw);
      for i:=1 to onum do
         begin
              readln(afile,xo,yo,phi,nvert);
              phi:=phi/convert;
              new(obstacle);
              obstacle .x:=xo;obstacle .y:=yo;obstacle .theta:=phi;
              formobject(xo,yo,phi);
              obstacle^.next:=vertex1;
              xo:=vertex1^.x;yo:=vertex1^.y;
              ox:=trunc((xo-100.0)*gra);oy:=trunc((yo-100.0)*gra);
              sgopen(i);
              beginp(ox,oy,1);
              fill(6);
              drawpolygon(obstacle);
              endp;
              psformobject(obstacle,margin);
              free(obstacle^.next);obstacle^.next:=vertex1;
              drawpolygon(obstacle);
              sgclose;
              gformobstacle;
              new(gobstacle[i]);
              gobstacle[i]^.x:=obstacle^.x;gobstacle[i]^.y:=obstacle^.y;
              gobstacle[i]^.next:=gvertex1;
              for k:=-tslices to tslices do appendobstacle(i,k);
              free(obstacle^.next);dispose(obstacle)
          end;
      readln(afile,factor);
      object^.x:=xf;object^.y:=yf;object^.theta:=fphi;
      drawpolygon(object);
      object^.x:=xi;object^.y:=yi;object^.theta:=iphi;
      drawpolygon(object);
      new(root);
      root^.x:=xi;root^.y:=yi;
      root^.son:=nil;
      root^.parent:=nil;
      root^.phi:=iphi;
      fp(root,xf,yf,fphi,xf,yf,fphi,0,leaf,feasible);
      if (feasible) then
        begin
             writeln('** path found **');readln;
             retrace
        end
      else abort;
      100:
      readln;for i:=1 to onum do sgdel(i)
end.
```

APPENDIX C

PROCEDURAL SKELETON OF THE CONFIGURATION SPACE ALGORITHM IMPLEMENTATION

```
program avoid obstacle,
        :
        procedure pchkint (xa, ya, xb, yb, real; gobs : gpoly;
                           K:integer;  var t1, t2; real);
```

(Checks for intersections of the line joining (xa, ya) and (xb, yb) on the plane parallel to the x-y plane at ϕ_k, with the k-th slice of the 3-D obstacle gobs. The points of intersections (x_i, y_i) are specified in terms of t_i where $x_i = xa + t_i(xb-xa)$, $y_i = ya + t_i(yb-ya)$)

```
        procedure gchkint [xa,ya,aphi, xb, yb, bphi:real; gobs:gpoly;
                            var t1, t2: real]
```

(checks for intersedtions of the line joining (xa,ya,aphi) and (xb,yb,bphi), with the 3-D obstacle gobs. The points are again given as t_i's).

```
        procedure fp(an: stree, xb, yb, bphi, xr, yr, rphi:real,
                     W:integer, var bn:stree, var fea: boolean),
```

(a_n is the last node of the path list. A path is to be found from point corresponding to a_n to (xb,yb,bphi). (xr,yr,rphi) is the midpoint of the segment of intersection in theearlier recursion step. W indicates whether the path from a_n to (xb,yb,bphi) is the former or the latter component of the earlier recursion step, in which the path has been decomposed into two. (xr,yr,rphi) and w together help deciding the acceptability of an intermediate point).

1) gchkint with all obstacles

2) if (no intersections) then

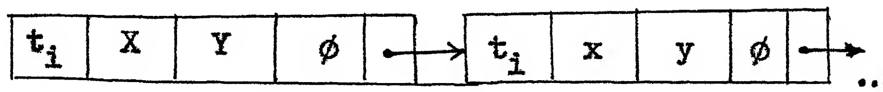
   (a) append b_n to pathlist after a_n

   (b) set fea:= true)

   else

   (a) for every allowable ϕ_k, pchkint for intermediate points on ϕ_k plane.

   (b) store all intermediate points in a list.



   (c) recursive call

   fp(a_n, x, y, ϕ, xp, yp, ϕp, 1, Cn, fea);

   fp(Cn, xb, yb, bphi, xp, yp, ϕp, 2, bn, fea);

Date Slip

This book is to be returned on the date last stamped.
